THE APPLICATION OF HYBRIDIZED GENETIC ALGORITHMS TO THE PROTEIN FOLDING PROBLEM

THESIS

Robert L. Gaulke
Captain, USAF

AFIT/GCS/ENG/95D-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/95D-03

THE APPLICATION OF HYBRIDIZED GENETIC ALGORITHMS TO THE PROTEIN FOLDING PROBLEM

THESIS

Robert L. Gaulke
Captain, USAF

AFIT/GCS/ENG/95D-03

AFIT/GCS/ENG/95D-03

The Application of Hybridized Genetic Algorithms to the Protein Folding Problem

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Robert L. Gaulke, B.S.C.S.
Captain, USAF

December 19, 1995

Acknowledgments

I would like to thank all those who helped me in this research. My first thanks go to my thesis advisor, Dr. Gary Lamont. I thank him for his patience and endurance of this strong-willed student. I also thank my sponsor, Dr. Ruth Pachter, for spending her time in the meetings and for setting up the Wright Labs presentation which got me on my way. I would also like to thank the members of my thesis committee, Major Gregg Gunsch and Dr. Eugene Santos for their time and teaching in the Artificial Intelligence sequence coursework and for their input for this thesis effort.

I am deeply indebted to Larry Merkle and George Gates for their sacrifice in time and energy on my behalf. They spent many hours in front of blackboards and monitors explaining algorithmic and molecular concepts to me. I could not have completed this research without the insight gained from our face-to-face and e-mail discussions. Also, I thank Larry again for not always just giving me the answer, but for making me *think*.

I also thank the following for insight offered and for keeping my sense of humor intact: Kevin Anchor, Chuck Beem, Randy Broussard, Jerry Cochran, Darin Goosby, Shawn Hannan, Vince Hibdon, Don Hill, Dave Kaneshiro, Pedro Lima, Conrad Masshardt, Shawn Northrop, Ding-Yuan "Steven" Sheu, Tom Rathbun, Ed Williams, and Oswaldo Zanelli.

Lastly, I wish to thank my parents (both sets!) and my brother and sister for their love and support. Most importantly, my love and thanks go to my wife, Lisa, for her understanding, support, and for *never* believing me when I said, "I'm just playing solitaire while it compiles."

Table of Contents

Table of Figures

Abstract

The protein folding problem consists of attempting to determine the native conformation of a protein given its primary structure. This study examines various methods of hybridizing a genetic algorithm implementation in order to minimize an energy function and predict the conformation (structure) of [Met]-enkephalin.

Genetic Algorithms are semi-optimal algorithms designed to explore and exploit a search space. The genetic algorithm uses selection, recombination, and mutation operators on populations of strings which represent possible solutions to the given problem.

One step in solving the protein folding problem is the design of efficient energy minimization techniques. A conjugate gradient minimization technique is described and tested with different replacement frequencies. Baldwinian, Lamarckian, and probabilistic Lamarckian evolution are all tested.

Another extension of simple genetic algorithms can be accomplished with niching. Niching works by de-emphasizing solutions based on their proximity to other solutions in the space. Several variations of niching are tested.

Experiments are conducted to determine the benefits of each hybridization technique versus each other and versus the genetic algorithm by itself. The experiments are geared toward trying to find the lowest possible energy and hence the minimum conformation of [Met]-enkephalin. In the experiments, probabilistic Lamarckian strategies were successful in achieving energies below that of the published minimum in QUANTA.

The Application of Hybridized Genetic Algorithms to the Protein Folding Problem

I. Introduction

Since the influx of computers into our culture began, we have been steadily increasing our reliance on their power to solve problems efficiently and accurately. As we attempt to solve more difficult problems, we have to work with larger search-space dimensions. These larger search space dimensions can result in lengthy solution times. So, we have to find ways beyond hardware (e.g. more powerful CPU chips) to speed up the process of finding solutions to our problems.

At the Air Force Institute of Technology (AFIT), we are studying two techniques for solving large problems quickly. First, there has been research in the areas of semi-optimal algorithms (5, 17, 43). As their name implies, semi-optimal algorithms solve problems by finding user-defined *good* solutions which are not necessarily (and frequently not) optimal. Semi-optimal algorithms offer a relatively fast way to get a quality solution to an extremely complex problem. (17, 34) Also, there has been ongoing research in the potential gains of Parallel and Distributed Computing (4, 17). Parallel computing involves efficiently dividing tasks to be simultaneously executed on multiple processors in order to realize some speedup versus execution on a single processor (34). Distributed computing involves dividing tasks among several systems (e.g. a group of workstations) to realize some speedup (48).

This particular thesis effort focuses on the **first** technique which is to apply semi-optimal algorithmic strategies in effort to solve the *protein folding problem*. This problem is to predict the three-dimensional structure of a protein given the primary sequence of amino acids that make up that protein. There are potentially a large number of bonded atoms in a protein and so there are many possible ways to arrange those atoms of a protein (and hence vary the layout of that protein). This plethora of protein arrangements can produce a search space so large that traditional searching methods (e.g. branch and bound, enumerative) can not be used. (5,7,17)

Background

This section provides a background on algorithmic complexity. Then, this section briefly discusses evolutionary algorithms focusing on genetic algorithms. This section closes with a short discourse on the protein folding problem.

Algorithmic Complexity

Many optimization problems involve a branch and bound search which can lead to traversing the entire solution space to be guaranteed to find the best solution. However, this type of search would experience exponential growth in execution time. (9, 35, 50, 59) This growth severely limits our ability to solve practical problems of any significant size. We want to therefore utilize techniques that allow us to search these larger problems. Two possible techniques are parallel/distributed computing (which is discussed in

Appendix A for the benefit of other researchers) and the use of stochastic search algorithms such as genetic algorithms (the concern of this thesis).

Genetic Algorithms

The family of *evolutionary algorithms* are made up of evolutionary strategies, evolutionary programming, and genetic algorithms. Evolutionary algorithms use a number of operators such as selection (reproduction), crossover, and mutation (these operators are discussed in detail in Chapter II). *Evolutionary strategies* (ESs), employed primarily in Europe, use the selection and mutation operators. ESs use a high rate of mutation on real-value encodings. Evolutionary Programming (EP), used primarily in the United States, also uses selection and mutation on real-value encodings. Genetic Algorithms use selection, a high probability of crossover, and a low probability of mutation.

Genetic algorithms are modeled on natural selection and genetics in that they simulate the *survival of the fittest* theory. It is important to note that a genetic algorithm does not necessarily find the optimal solution but it finds a *good* solution (hence the term *semi-optimal*). A genetic algorithm is of polynomial time complexity with a finite space requirement which is determined by the population size. So, the genetic algorithm enables us to obtain a good solution of a problem of exponential complexity in polynomial time. Genetic Algorithms were developed in an attempt to create robust, semi-optimal search algorithms that would be applicable to a wide variety of problems. However, a major shortcoming of GAs is premature convergence to local optima. In other words, the

algorithm tends to get hung on a local optimum and returns it as a solution instead of a globally better solution. (4, 5, 17, 19, 23, 26, 35, 46)

Genetic Algorithms are easily parallelized. One approach puts multiple copies of the same program on each processor, starts their execution with different seeds for the random number generators, and selects the best solution after all processors have finished. Another approach (referred to as the *island model*) is where the population is divided up into subpopulations which are grouped on individual processors which run independent genetic algorithms. This results in little communications overhead but at a possible sacrifice in solution quality. The execution time of a genetic algorithm is typically dominated by the calculation of the fitness function. This function is problem dependent, but is usually of polynomial time complexity.(17, 21, 19) In part 2, the Literature Review, we discuss the details of **how** a genetic algorithm works (See Appendix A for a further discussion on Parallel/Distributed Computing).

The Protein Folding Problem

The protein folding problem is classified as a *Grand Challenge* problem (47). The protein folding problem consists of trying to map out the secondary and tertiary structure (conformation) of a protein molecule given its primary structure. The primary structure of a protein is the sequence or chain of amino acid residues. The secondary structure represents the 3-dimensional arrangement of amino-acid residues within the molecule (e.g. α-helix or β-sheet). The tertiary structure defines the molecule in terms of the relative

position of its bonded atoms. The purpose behind finding the structure mappings of a protein is that properties and functions of the molecule may be determined by its structure. So, if a relatively quick yet correct method of mapping out the structure of proteins can be formulated, we can greatly speed the development of industrial, pharmaceutical, and military applications. (5, 7, 17, 43, 57)

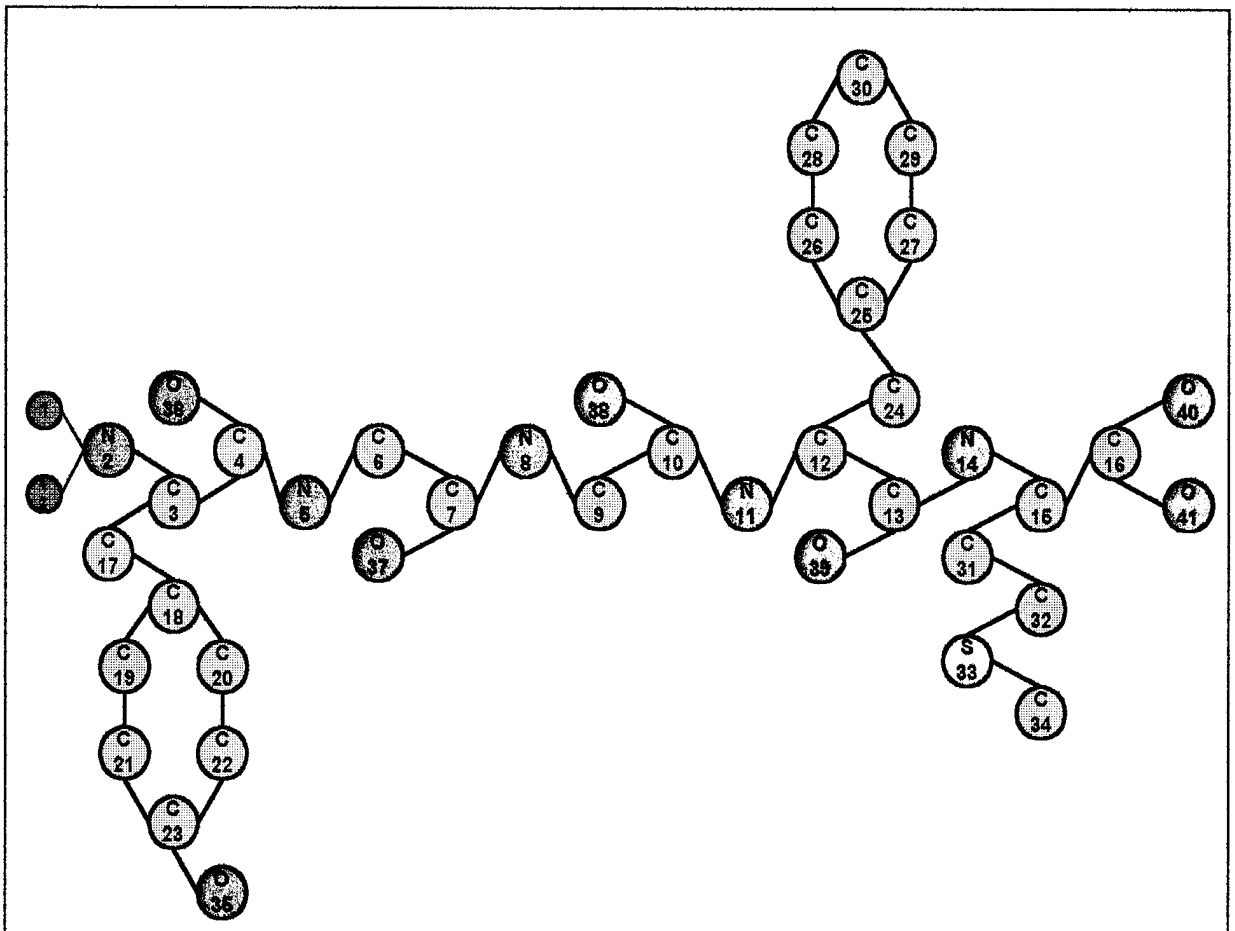


Figure 1: An extended conformation of [Met]-enkephalin (17)

The protein used in most of the AFIT studies is [Met]-enkephalin (see Figure 1). It is a relatively small and simple protein (polypeptide) defined by the five-amino-acid

sequence *Tyr-Gly-Gly-Phe-Met*. It is principally composed of carbon (C), oxygen (O), and nitrogen (N) atoms. The two principal factors influencing the selection of this particular protein for study are: [1] its unique and compact natural, biological state (*native conformation*) is known; [2] other researchers have used energy minimization to predict its tertiary structure. (5, 17)

While current experimental techniques allow us to decode the primary structure of a protein with little effort, predicting the tertiary structure of a protein is extremely difficult. Nuclear magnetic resonance and X-ray crystallography are laboratory techniques for determining the three-dimensional conformation of a protein. However, these approaches can expend as much as two years of laboratory effort to find the tertiary structure of a single protein and are not always possible! (16, 57)

The solution to the Protein Folding problem can be modeled with an energy minimization approach. Energy minimization is a basic technique for predicting the tertiary structure of a protein using one of the following approximation methods: [1] *ab initio* methods: use quantum mechanical calculations to determine the energy exactly; [2] use *semi-empirical* methods that neglect some of the non-dominating energy terms; [3] use *force-field* methods which only account for pairwise energy interactions between atoms. Calculating a single energy value for these methods is of time-complexity $O(n^5)$, $O(n^4)$, and $O(n^2)$ respectively (where n is the number of particles - frequently atoms). (5, 7, 17, 39) This need for faster methods is the driving force behind the use of genetic algorithms.

Problem Statement

The challenge of solving the Protein Folding Problem is to find a method of predicting the 3-dimensional shape of a protein given its defining sequence of amino-acids. An enumerative search of the entire solution space for even the smallest proteins would consume more time than the estimated age of the universe on today's supercomputers. Recent AFIT research (17) has indicated that parallel genetic algorithms are feasible for predicting the tertiary structure of the pentapeptide [Met]-enkephalin. Some goals for this investigation are to continue to improve the performance of the simple genetic algorithm and to continue to evaluate the feasibility of applying parallelized evolutionary algorithms to predict the tertiary structure of more complex molecules. While attempting to accomplish these goals, the major objectives of this effort are:

a) make improvements to the simple genetic algorithm design and implementation

b) investigate the use of hybrid optimization techniques such as local minimization to improve genetic algorithm efficiency and effectiveness. For example, conjugate gradient methods (deterministic) and simulated annealing (probabilistic) are two potential local minimization techniques.

c) investigate the use of niching strategies to improve genetic algorithm efficiency and effectiveness.

Rationale

Why is the Protein Folding problem important? Solving the protein folding problem implies the ability to efficiently and reliably predict the tertiary structure of any protein once given the primary structure of that protein. Knowing the function of the various proteins present in our own bodies could lead to many new medical and scientific breakthroughs such as preventing or curing disease, repairing genetic disorders or birth defects, and developing disease or pest resistant strains of plants! The solution of the protein folding problem is also significant because it could provide insight into its complementary problem, which is that given a particular function that we desire a protein to perform, what is the tertiary structure that performs that function? Then, how do we construct that protein, or what is the primary structure we should build? The solution of this complementary problem would allow biochemists and computational scientists to design new polypeptides with a single, specific purpose. (5, 17) Moreover, there are variety of military applications including the possible development of a photosensitive protein film to be used on protective goggles for pilots.

Methodology

There are a number of activities or tasks that make up this research effort. The following subsections identify and define the major tasks that are to be accomplished in approaching this research effort.

Literature Review

This continuing review is used to establish foundations of current knowledge in the applicable fields of study. The principle review areas along with references are:

a) genetic algorithms (4, 5, 13, 17, 19-21, 23-26, 29, 31-33, 35, 43, 46, 57)
b) protein folding problem (5, 7, 16, 17, 19, 20, 21, 32, 33, 43)
c) hybridization techniques (1, 12, 28, 37, 40, 46, 49, 51, 55, 56)

Software Review

This involves the study and comprehension of the programs contained in the AFIT Genetic Algorithm Toolkit as well as any code obtained from other sources for possible integration. (30, 36, 51)

Algorithm Modifications/Extensions

As problem areas are discovered, the design is to be modified accordingly. The principle extension is the addition of Local Minimization techniques to the algorithm and implementation.

Implementation Modifications/Extensions

After modifications have been made to the algorithm, the implementation is to be appropriately modified. For instance, following the extension of local minimization

techniques in the algorithm, the actual implementation is to be modified (or extended) to reflect the change in the algorithm.

Experiment Design and Implementation

After the reviewing the software, reviewing the literature, and modifying the implementation , experiments are designed using the modified code. The experiments are designed to generate useful data so that this effort builds upon the work completed by previous AFIT students.

Analysis

The final step involves analyzing the data generated from performing the experiments, drawing comparisons between the experiments' data, evaluating what has been accomplished, and making recommendations on where future research should be focused. Particular emphasis is to be placed on the comparison of the results from the implementation *as-is* and the results of the implementations that use the various hybridization techniques.

Summary

Large, complex optimization problems require the use of suitable semi-optimal algorithms that trade some amount of solution quality for substantially reduced execution times. This thesis effort compares hybrid and standard genetic algorithm techniques for

efficiency and effectiveness in finding solutions. It also takes the recent research further by analyzing the feasibility of using AFIT's genetic toolkit software to determine the tertiary structure of larger, more complex proteins.

This chapter has outlined the general problem, described the main components, and rationalized the need to expend research effort on genetic algorithms and the protein folding problem. Chapter II discusses the protein folding problem while Chapter III details genetic algorithms. Chapter IV analyzes several hybridization techniques with attention given to the benefits of each to genetic algorithm implementations. Chapter V discusses the design and implementation of the experiments. Chapter VI presents the experimental values and the experimental data which is evaluated to establish the conclusions of this research. Finally, Chapter VII draws overall conclusions and presents some recommendations concerning future efforts in genetic algorithms and their application to the protein folding problem.

II. Literature Review - The Protein Folding Problem

This chapter summarizes current knowledge of the Protein Folding Problem in order to establish a foundation for this thesis effort. The discussion is to first focus on protein structures followed by a look at current laboratory methods. Then, this chapter defines some terms of molecular geometry. Finally, the problem search and solution spaces are examined.

The Protein Folding Problem

Proteins are very common molecular structures. Several types exist: fibrous, membrane, and globular. Fibrous proteins make up the structural components in the human body. Membrane proteins control the flow of material across cellular boundaries. Enzymes which control biochemical reactions in cells (and thus are of interest to us) are globular proteins. (5, 7, 17)

A protein's primary structure is a sequence of amino acids. Thanks to modern technology, we can use computers to sequence a protein to rapidly determine its primary structure. As stated in Chapter I, the Protein Folding Problem consists of trying to map out the tertiary structure of a protein molecule given its *primary* structure. The primary structure is the chain of amino acids. Due to charges of each amino acid, the chain folds into a *secondary* structure. The three main characterizations of the secondary structure are the α-helix, the β-sheet, and the looped domain. Based on the net free-energy of the molecule, the secondary structure folds again to form the *tertiary* structure of a protein.

Tertiary structures are then used in the cellular functions described. The tertiary structure defines the exact shape of the entire molecule. In other words, the tertiary structure is the actual layout of the atoms including the angles of the bonds between them. In searching for possible three-dimensional atomic arrangements (the tertiary structure) of a protein, we are looking for a stable protein which has a low energy. Because we are trying to find the best layout (*conformation*) of a particular protein by varying the dihedral angles (see Figure 2), this produces the folding effect of the protein and thus the name, the *Protein **Folding** Problem*. The purpose behind finding the structure mappings of a protein is to the determine properties and functions of that molecule. So, if a relatively quick, yet effective method of mapping out the structure of proteins can be formulated, we can greatly speed the development of pharmaceutical and military applications. (5, 7, 16, 17, 44, 57)

Current laboratory methods of determining the tertiary structure are slow and tedious. X-ray crystallography involves striking a protein crystal with a fine beam of X-rays which creates a diffraction image on a photographic plate. The diffraction is proportional to the number of extranuclear electrons in each atom. A series of two-dimensional images are then used to calculate a three-dimensional image. The researcher must be able to grow a well-ordered, ninety-seven percent pure protein crystal (the growth alone can take months) and then be able to dehydrate the crystal for maximum diffraction resolution. (57)

Nuclear magnetic resonance (NMR) techniques are based on plots of characteristic signals of hydrogen atom interference. These signals are used to identify amino acids and determine interatomic distances that are then used to reconstruct protein structure using computer graphics. Both approaches need high concentrations of proteins to make accurate determinations. (16) However, these approaches can expend more than **two years** of laboratory effort to find the tertiary structure of a single protein and are sometimes not even possible.

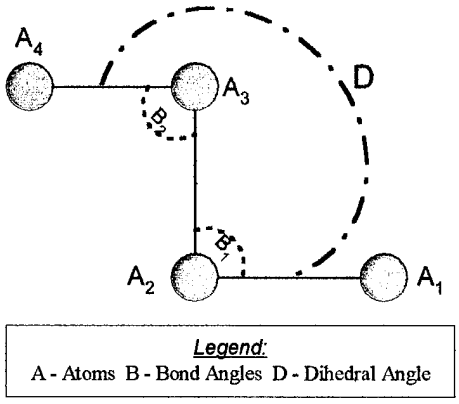


Figure 2: Example Molecule Geometry

For our discussion about molecular geometry, see Figure 2. We have four atoms (A_1, A_2, A_3, A_4) connected by bonds. There are two *bond angles* (B_1 and B_2). B_1 is the angle formed by the A_1A_2 bond and the A_2A_3 bond and B_2 is the angle formed by the A_2A_3 bond and the A_3A_4 bond. There is only one *dihedral angle* (D). It is the angle formed between the A_1A_2 bond and the A_3A_4 bond. We can vary the dihedral angle, D, by twisting or *folding* our structure about the A_2A_3 bond. Through this kind of folding we

can alter the shape of our simple structure (i.e. produce many conformations). In a more complex structure such as [Met]-enkephalin (see Figure 1 in Chapter I), where there are twenty-four dihedral angles, there are many possible conformations. [Met]-enkephalin is actually a rather simple protein structure with a relatively low number of dihedrals. Thus, larger proteins can present an enormous search space of conformations. (4, 17, 43)

Minimizing the energy function of a protein is a complex undertaking (see Figure 3 for the energy function). Factors contributing to the complexity are the large search space, computational intensity of the determination of an individual's energy, and the existence of many local minima. How big is the search space? Consider that a

$$
\begin{aligned}
E = \quad & \sum_{(i,j)\in B} \left(K_{rij}\,(r_{ij} - r_{eq})^2\right) + \\
& \sum_{(i,j,k)\in A} \left(K_{\Theta ijk}\,(\Theta_{ijk} - \Theta_{eq})^2\right) + \\
& \sum_{(i,j,k,l)\in D} \left(K_{\Phi ijkl}\,(1 + \cos(n_{ijkl}\Phi_{ijkl} - \gamma_{ijkl}))\right) + \\
& \sum_{(i,j)\in N} \left((A_{ij}/r_{ij})^{12} - (B_{ij}/r_{ij})^{6} + q_i q_j/4\pi\varepsilon r_{ij}\right)
\end{aligned}
$$

where: K_{rij}, $K_{\Theta ijk}$, $K_{\Phi ijkl}$, r_{eq}, Θ_{eq}, n_{ijkl}, γ_{ijkl}, A_{ij}, and B_{ij} are empirical constants
B - bonded atoms, A - atoms forming bond angles, D - atoms forming dihedral angles,
N - non-bonded atoms (atoms with more than 3 bonds separating them)

Figure 3: CHARMm energy model (38)

protein can have up to hundreds of amino acids. Thus, a protein can have a tremendous number of atoms (sometimes hundreds of thousands). Moreover, a protein has 3n-6 degrees of freedom (where n is number of atoms). In a protein with just fifty residues (having twenty atoms per residue), we would be dealing with a system of equations with 3*(20*50) - 6 = 2994 variables! Because we can discretize each dimension of the search

space to some domain of values (d), our search space has a complexity of $||d||^{3n-6}$. Moreover, since the bond angles and lengths are relatively stable, the dihedral angles mostly determine the tertiary structure. So, the search space can be reduced to $||d||^n$ where d is the number of discrete dihedrals and n is the number of the independently variable dihedrals. However, what if we had just ten independent dihedral angles and dihedral angles discretized over twenty degree increments in a range of 0 to 360 degrees. Our search space would be to the order of 10^{18} (# of dihedrals $^{\text{\# of degrees / \# of increments}}$). It would take about eleven days to search this relatively *small* search space on a teraflop (capable of one trillion floating point operations per second) computer which is capable of one point evaluation per clock cycle. Now, imagine how long it would take to search a protein with 100 dihedrals! Thus, we need faster methods of calculating the tertiary structure from a protein sequence. (5, 17, 44, 53)

Summary

This chapter has presented a discussion of the Protein Folding Problem. It is a very large and complex problem that is not easily solved with laboratory techniques. So, the use of computers with efficient algorithms is justified. Solving this problem could pave the way for developments in pharmaceuticals and military applications. The next section is on genetic algorithms. It is important to consider that genetic algorithms may offer a method for generating good solutions to the protein folding problem but, by nature, can not ***consistently solve it***.

III. Literature Review - Genetic Algorithms

This chapter summarizes current knowledge of genetic algorithms in order to establish a foundation for this thesis effort. After a brief discussion on some of the primary people working in the field of genetic algorithms, this chapter details how a genetic algorithm works. Next, there is an analysis of the simple genetic algorithm including discussions on genetic operators. Then, there is a section addressing the fundamental theorem of genetic algorithms. This chapter also discusses the messy and fast messy genetic algorithms with a look at advantages of disadvantages of using each.

Genetic Algorithms

Background

The foundations of genetic algorithms can be traced to a University of Michigan researcher, John Holland, and to one of his early students, Kenneth DeJong. Genetic algorithms were first proposed in *Adaptation in Natural and Artificial Systems* (1975), by Holland. There, he established the mathematical basis for genetic algorithms. DeJong, in his dissertation *An Analysis of the Behavior of a Class of Genetic Adaptive Systems* (1975), took Holland's work a step further by applying genetic algorithms to functional optimization problems. (13, 17, 23, 31, 44)

Other principal contributors to genetic algorithm research are David Goldberg, Zbigniew Michalewicz, and John Grefenstette. *Goldberg*, who is also a Michigan alumnus, started with a dissertation that investigated the use of genetic algorithms to

control gas-pipeline transmission (which earned him a NSF Presidential Young Investigator Award in 1985). This and his subsequent work through the rest of the eighties culminated in Genetic Algorithms in Search, Optimization, and Machine Learning (1989). This textbook is used as a basic *handbook* for both fledgling and experienced genetic algorithm researchers alike. Goldberg is one of the most published individuals of the genetic algorithm field. Researchers of all levels also depend on the textbook by *Michalewicz*. His book, titled Genetic Algorithms + Data Structures = Evolution Programs (1992), introduces and examines genetic algorithms and their applicability to artificial intelligence and optimization problems. While he has worked on genetic algorithm parameter sets and machine learning, *Grefenstette*'s best known contribution is GENESIS. GENESIS is a genetic algorithm implementation used by many researchers (including those here at AFIT) as a basic workbench. (17, 23, 30, 46)

Using genetic algorithms involves searching through a space of potential solutions which necessitates exploring the solution space and taking advantage of the best solutions generated. While neglecting exploration of the search space, Hillclimbing takes advantage of the best solution for possible improvement. However, a random search explores the search space while not using any knowledge of promising areas to its advantage. Michalewicz states that:

> "Genetic Algorithms are a class of general purpose (domain independent) search methods which strike a remarkable balance between exploration and exploitation of the search space." (46)

Genetic algorithms work by manipulating populations of *strings* (or *chromosomes*). These strings are possible solutions encoded usually with ones and zeros (a series of *genes*) representing Boolean conditions. For instance, say the problem being solved is the classic Knapsack Problem where we wish to maximize the value of the weight we can carry in our knapsack. The knapsack problem can be represented with 0/1 notation in that (1) you have an item or (0) you do not. So, the strings would represent possible combinations of items in our knapsack. Strings are selected for the next *generation* based on their *fitness*. Our knapsack fitness function, would be based on the items' value and weight. (5, 10, 17, 19, 23, 29, 43, 46)

Genetic algorithms continue to generate populations for a defined number of generations after which time the current best string is used as the solution to the problem. The execution time of a genetic algorithm is typically dominated by the calculation of the fitness function. This function is problem dependent, but is usually of polynomial time complexity. Genetic algorithms can be classified into two main types: the *simple* (or *standard*) genetic algorithm and the *messy* genetic algorithm (see Appendix B). (5, 17, 19, 23, 43, 46)

The Simple Genetic Algorithm

How do simple genetic algorithms work? Simple genetic algorithms keep uniform length strings and perform three basic operations on those strings in the population: *selection*, *crossover*, and *mutation*. Refer to Figure 4 for the general structure of the algorithm.

First, the population is initialized. The population size is 2^l (where l is the length of a string). (17, 23) This is because having strings with l binary digits each means that it takes 2^l different strings to represent all possible values. For example, if our strings have four digits (1's and 0's), then there are sixteen (2^4) possible strings that can be formed.

```
initialize population
for i = 1 to max_number_of_generations
     evaluate fitness
     for j = 1 to population_size
          selection
          crossover
          mutation
          evaluate fitness
     end loop
end loop
```

Figure 4: Simple Genetic Algorithm

The selection (sometimes referred to as *reproduction*) operation does just what the name implies - it selects members of the current population or *generation* to carry over into the next generation. Simply stated, "let's give more copies to better guys" (27). The selection of strings is based on their *fitness*. The fitness can be defined as an enumeration of goodness or utility that the algorithm is to maximize. In our case, the fitness is the potential energy of the protein. However, by itself selection is not very useful. In fact, if we were to only do selection steps over and over, we would likely wind up with many copies of the best solution of the first generation. (17, 23, 27)

There are several types of selection operators. First, there is the Roulette Wheel selector. The roulette wheel selector (which is commonly used in simple genetic algorithms) assigns each string to a section of a wheel proportionate to the ratio of the

string's fitness and the average fitness of the population. So, the roulette wheel is an example of a *fitness proportionate* selector. Figure 5 demonstrates how different

String	Fitness	Ratio
S1	15	1.5
S2	5	0.5
S3	10	1
S4	10	1
mean	10	

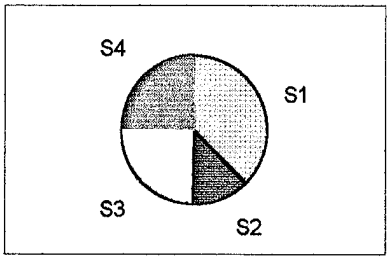


Figure 5: Roulette Wheel Selection

fitnesses provide different proportions of the wheel. Thus, S1 is three times as likely to be selected as S2 while both S3 and S4 are each twice as likely to be selected as S2. (17, 23, 43, 44)

Another type of selection operator is the *tournament* selector. It iteratively selects random pairs of strings which satisfy a thresholding criteria. It then compares the strings and picks the better one. Poorer strings (which would survive into later generations in the roulette wheel selector) are eventually gleaned from the population. So, this operator is labeled fitness *disproportionate*. (17, 23, 43, 44)

The crossover (also referred to as *recombination*) operator involves mating two strings and hence mixing their characteristics. This is accomplished by choosing a random crossover point and swapping the portions of the strings after the crossover point. In Figure 6, two members of the present generation called P1 and P2 mate to form two members of the next generation called N1 and N2 (random crossover point happens to

occur in the middle of the strings). What if we were to go from generation to generation, just doing crossover? The result would be a randomly mixed population whose probability distribution would match what we get from just shuffling the bits we had initially at random. (5, 17, 23, 24, 27, 43, 46)

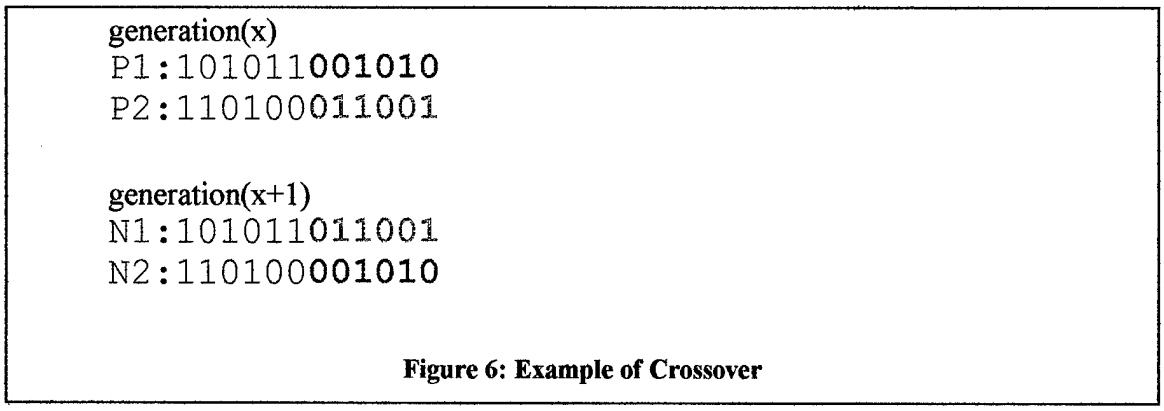


Figure 6: Example of Crossover

The mutation operator (see Figure 7) simulates a sudden, random change in a string. Mutation occurs at a random point in a string and the bit value is changed (1 to 0 or 0 to 1). This causes the solution to randomly explore the solution space. Mutation occurs much less frequently than crossover. In Figure 7, a random point is chosen on P1 and N1 is formed by flipping that bit. (5, 17, 23, 24, 43, 46)

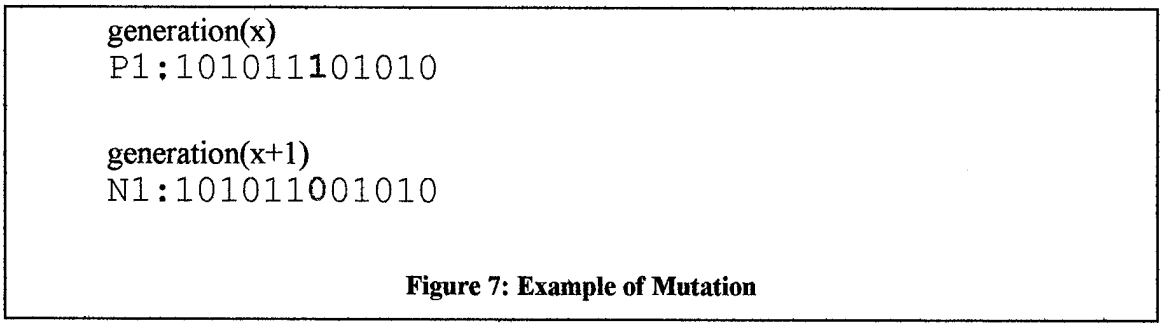


Figure 7: Example of Mutation

The algorithm steps through these three operations repeatedly until some stopping criteria is met (max_number_of_generations in Figure 3). The combined effects of the selection operator (optimizer) and the recombination operator (diversifier) create the robust searching capability of the genetic algorithm while the mutation operator can help to aim the algorithm toward still other parts of the search space. Note that in all the above examples, binary strings were chosen for simplicity. However, real-valued strings, Lisp codes, and even assembly instructions may be used as well. Because genetic algorithms are loosely based on natural evolution, many of the terms associated with natural evolution are used interchangeably with the terms created specifically for genetic algorithms. (4, 5, 17, 19, 23, 24, 27, 29, 43, 46)

Fundamental Theorem of Genetic Algorithms

So, *what* makes the genetic algorithm work? Holland, in his book, discussed a theorem dealing with the probability of a string's survival from one generation to the next. This later became known as the Fundamental Theorem of Genetic Algorithms or the Schema Theorem. Before discussing the theorem, we need to define some terms. A *schema* is a pattern or template used to describe sets of strings with the same values at certain positions. The positions having different values are indicated by the *don't care* symbol (*). For example, 1*1 defines the set of strings {101, 111} while 1**0 defines the set of strings {1000, 1010, 1100, 1110}. Two values associated with a particular schema H, are the *defining length* (δ(H)) and the *order* (o(H)). The defining length indicates the number of positions between the first specified value of the string and the last specified

value of the string. For example, 1**0* has a defining length of three (4-1) while *0***** has a defining length of zero (2-2). The order of a schema indicates the total number of specified positions. So, 1**0* has an order of two while *0***** has an order of one. (4, 17, 23, 43, 44)

When crossover occurs within the defining length of the schema, it is possible (but not certain) that the schema can be disrupted. So, the probability of a schema's survival (p_s) under crossover (which itself has a probability of p_c) is:

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1}$$

Figure 8: Probability of schema survival under crossover

Mutation can also disrupt the schema. So, the probability of a schema's survival (p_{sm}) under mutation (which itself has a probability of p_m) is:

$$p_{sm} \approx 1 - o(H)p_m, \; p_m << 1$$

Figure 9: Probability of schema survival under mutation

Let f(H) be the average fitness of a string matching schema (H), and $\bar{f}$ be the average fitness of the population. Moreover, suppose that the number of schema-matching strings in a population at time (t) is m(H,t). Then, the reproduction operator has the effect of:

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}}$$

Figure 10: Effect of reproduction operator on the schemata

Considering the combined effects (omitting a few negligible terms) of reproduction, crossover, and mutation on a schema's survival, the Schema Theorem indicates the number of examples of a schema in the next generation:

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}}(p_s + p_{sm}) \approx m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right]$$

Figure 11: Schema Theorem

The Schema Theorem (in Figure 11) can be interpreted as saying "short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations." (23) In other words, small schemata that do not have very many specified positions but do have a good average fitness are more likely to survive and are therefore to be tested many times in later generations. (4, 17, 23, 43, 44, 46)

A setback of the simple genetic algorithm is the problem of *deception*. Deception is where a genetic algorithm selects locally optimal building blocks rather than globally optimal ones resulting in a premature convergence and an incorrect answer. In other words, short, low-order building blocks leading to suboptimal higher order building blocks causes deception. This is frequently the result of a function whose best points are surrounded by the worst, or in other words, a function with *isolated optima*. It can be argued than many optimization techniques would not perform in the case of a function

with local optima and that such functions occur rarely. Nonetheless, in order to combat the problem of deception, Goldberg devised another type of genetic algorithm — the *messy genetic algorithm*. Messy genetic algorithms (which are not used in this thesis effort) are discussed in Appendix B. (23, 46)

Summary

The various forms of genetic algorithms offer us different approaches to finding solutions to problems. However, as we start to deal with real-world problems (which often have a massive search space), whatever type of genetic algorithm we choose to work with is too slow. For example, the genetic algorithm can take several hours to find a good solution when attempting to minimize the energy of [Met]-enkephalin which is a relatively small protein. Therefore, we must find methods to be used in conjunction with genetic algorithms to make our searches more efficient and effective (21).

IV. Literature Review - Hybridization Techniques

This chapter summarizes current knowledge of hybridization techniques with emphasis placed on the possible benefits of combining them with a genetic implementation in order to solve the protein folding problem. First, this chapter addresses local minimization which includes a discussion of conjugate gradient techniques. Next, there is a discussion of simulated annealing. Then, the chapter analyzes possible minimization application strategies. This chapter closes with a discussion of niching.

Local Minimization

As an enhancement to our genetic algorithm, we wish to apply a local minimization step(s) that can improve upon the value returned by the genetic algorithm at that iteration. A genetic algorithm containing local minimization operators is sometimes referred to as a *Hybridized Genetic Algorithm* (HGA). Following a fitness evaluation, local minimization would move us closer to local minima among which (hopefully) is the global minimum.

Two categorizations of approaches that can be used to locally minimize a multivariable function are deterministic and probabilistic. A *deterministic* approach is characterized by using knowledge of the search space in making a decision. This is usually the result of some calculations which provide the knowledge of the search space. For example, a calculation may allow you to omit a section of (prune) the search space. On the other hand, a *probabilistic* approach does not make use of knowledge about the

search space. A probabilistic approach makes selections after altering the probabilities of accepting solutions. So, acceptance probabilities of inferior solutions would be dynamically lessened to decrease our chances of selecting those solutions. It is worth noting at this time that an *elitist* strategy (can be used in genetic algorithms) is similar to a probabilistic approach in that an elitist algorithm guarantees that the best solution is carried over to the next generation. So, in other words, it is probabilistic in that the probability of selecting the best solution equals one. The two methods detailed here are conjugate gradient techniques (a deterministic approach) and simulated annealing (a probabilistic approach). (1,4) Figure 17 shows how a local minimization step could fit into a simple genetic algorithm.

```
initialize population
for i = 1 to max_number_of_generations
     evaluate fitness
     for j = 1 to population_size
          Local Minimization step
          selection
          crossover
          mutation
          evaluate fitness
     end loop
end loop
```

Figure 12: Simple Genetic Algorithm with Local Minimization Step

Conjugate Gradient Techniques

To discuss the conjugate gradient technique, a brief mathematical review is necessary. What is a gradient? If the partial derivatives of $f(x,y,z)$ are defined at a particular point, then the gradient of f at that point is a vector of the corresponding first partial derivatives or symbolically:

$$\nabla f = \frac{\partial f}{\partial x} i + \frac{\partial f}{\partial y} j + \frac{\partial f}{\partial z} k$$

Figure 13: Gradient of a function

In other words, a gradient is a vector in the direction of the maximum directional derivative of a function. A conjugate direction is sometimes referred to as a *non-interfering* direction. In a sense, *conjugate* means perpendicular. It does not use just the latest vector but the best combination of all vectors reached. So, we are moving in a direction that is perpendicular to all preceding directions. (11, 15, 51, 55)

1. $A\mathbf{x} = \mathbf{b}$ (classic matrix equation)
2. Suppose the vectors $\mathbf{d}_1,\ldots,\mathbf{d}_n$ are A-orthogonal $\rightarrow (\mathbf{d}_i)^T A\mathbf{d}_j = 0$
3. Finding the components of $\mathbf{x}$, $\mathbf{x} = \alpha_1\mathbf{d}_1 + \alpha_2\mathbf{d}_2 + \ldots + \alpha_n\mathbf{d}_n$
4. Start at $\mathbf{x}_0 = 0$, where residual $\mathbf{b} - A\mathbf{x}$ is $\mathbf{r}_0 = \mathbf{b}$
5. Go in direction $\mathbf{d}_1$ of steepest descent, continue to point $\mathbf{x}_1 = \alpha_1\mathbf{d}_1$
6. Compute new residual $\mathbf{r}_1 = \mathbf{b} - A\mathbf{x}_1$
7. Move in direction conjugate to $\mathbf{r}_1$ which is $\mathbf{d}_2 = \mathbf{r}_1 + \beta_2\mathbf{d}_1$
8. Continue to point $\mathbf{x}_2 = \mathbf{x}_1 + \alpha_2\mathbf{d}_2$
9. Generalization of the cycle:
   - direction: $\mathbf{d}_j = \mathbf{r}_{j-1} + \beta_j\mathbf{d}_{j-1}$
   - next point: $\mathbf{x}_j = \mathbf{x}_{j-1} + \alpha_j\mathbf{d}_j$
   - residual: $\mathbf{r}_j = \mathbf{b} - A\mathbf{x}_j$
10. β_j is chosen to make $\mathbf{d}_j$ A-orthogonal to $\mathbf{d}_{j-1}$
11. At the nth step, $\mathbf{x}_n = \mathbf{x}_{n-1} + \alpha_n\mathbf{d}_n$ and so we have $\mathbf{x}$.

Figure 14: Matrix reduction with conjugate gradient technique (55)

In principle, to minimize a function with a conjugate gradient technique, we need to start by moving in a direction opposite the gradient (the direction can be referred to as a *residual* and is equal to the gradient with minus sign). Next, we minimize by stepping along the function in a direction that is conjugate to the residual (this is calculated as a new residual). Next, we step in a direction that is conjugate to *both* our initial heading and its conjugate. Then, we step in a direction that is conjugate to our first heading, its

conjugate, and their conjugate and so on. See Figure 20 for a matrix representation of calculating the conjugate gradient. (51, 55)

Simulated Annealing

Simulated annealing (see Figure 21) is a probabilistic algorithm based on thermodynamic principles relating to the way that liquids freeze and crystallize and metals cool and anneal. If a liquid is cooled slowly, the atoms can line themselves up and form a

```
Select an initial state i∈E
Select an initial temperature T > 0
Set t = 0
Repeat
        Set n = 0
        Repeat
                Generate state j which is a neighbor of i
                Calculate δ = C(j) - C(i)
                If δ < 0   Then i = j
                Else if random(0,1) ≤ exp(-δ/T)   Then i = j
                Increment n
        Until n = N(t)
        t = t +1
        T = T(t)
Until stopping criterion is true
```

Where:
E is the set of possible configurations (search space)
t is the temperature change counter
n is a repetition counter
C is a cost function that assigns a real number to each member of **E**
δ is the change in cost associated with moving from one state to another

Figure 15: Simulated Annealing Algorithm (37)

completely ordered crystal. This crystal is the minimum energy state for the system. The algorithm simulates this annealing process in solving optimization problems. It has also

been applied to problems in VLSI design, learning, artificial neural networks, and artificial vision. (37, 40, 46, 60)

Local Minimization Application Strategies

There are two different strategies or approaches to applying local minimization to evolutionary algorithms. First, *Lamarckian* evolution uses local search to improve the current population. It also encodes those improvements onto the strings to be processed for the next generation. On the other hand, the *Baldwinian* approach involves the combination of learning with evolution. This is accomplished by transferring the improved fitness value (from the local minimization step) to the individual without coding the improvements back onto the string. This simulates the lifetime learning of an individual. The Lamarckian approach tends to converge much faster but has a greater probability of missing the global optimum by converging to some local optimum instead. The question of which approach is better seems to be application-dependent. (58)

There are a number of ways to apply these approaches. The obvious technique is to perform a local minimization step every generation within the genetic algorithm. This application technique seems appropriate since we would be able to apply extra optimization to each member of the population at every genetic algorithm generation. The principal drawback is the massive amount of computation (repeatedly calculating the partial derivatives and gradients) involved with each local minimization step and then the resulting increase in execution time. Also, another drawback is that the application of

local minimization in every step could narrow the search space too rapidly and thus cause the global minimum (or maximum) to be missed. These two problems point to the need for an approach where a local minimization step would be applied only once in awhile. This kind of approach would still take advantage of the explorative nature of the genetic algorithm while reaping the additional exploitative benefits of local minimization. There are several possible strategies. One way is to apply a local minimization step every x generations of the genetic algorithm (where x is 5, 10, 20, or whatever you want). Another possibility would be to have a local minimization step whenever a mutation occurs. The current AFIT implementation uses a three tenths of a percent mutation probability so local minimization steps would be relatively infrequent with this approach. Finally, another strategy would be Orvosh and Davis's *five percent rule*. They propose to arbitrarily replace five percent of the strings in every generation by re-encoding them with results found by a local minimization step. Their work showed the *five percent rule* was more effective than either always or never replacing the repaired strings for the applications they were solving. This replacement strategy could be varied by replacing ten or fifteen percent of the strings. The arbitrary replacement of strings can also be referred to as *probabilistic Lamarckian replacement*. (45, 49, 58)

Niching

Another enhancement that can be applied to genetic algorithms is the concept of niching. Niching takes its concept from nature in that different species tend to exploit separate niches (sets of environmental features) in which other organisms have little or no interest rather than competing directly for the same resource. From an algorithmic point

of view, we have a solution space made up of a number of peaks and valleys. The genetic algorithm tends to concentrate its solutions around a certain peak. The idea of niching is to spread the genetic algorithm's population around to other peaks by de-emphasizing a member's fitness based on the proximity of other population members. There are several niching schemes such as *crowding* and *sharing*. (12, 28)

In crowding, we replace strings based on their similarity with other strings in an overlapping population. Stepping through generations of the GA, we randomly draw a subpopulation of crowding factor (CF) members. Then, we compare an individual to each string of the CF and replace the most similar string (based on a bit similarity count). As we progress to later generations, one or more species should establish a foothold in the population resulting in more strings being similar to each other. Then, by replacing similar strings, we can help maintain diversity and allow room for more species. (12, 28)

In sharing, we reduce a member's fitness based on that member's nearness to other members. In other words, a large cluster of individuals results in a large reduction in fitness for each while a solitary individual's fitness remains relatively unaffected. There are two forms of sharing - genotypic and phenotypic sharing. In genotypic sharing, we use sharing based on genetic proximity - the hamming distance between strings (number of different alleles). We use phenotypic sharing when the proximity is defined in the decoded parameter space. (12, 28)

Experiments run by Deb and Goldberg (12) suggest that for functions with unequal peaks that genotypic sharing sometimes is unable to maintain stable subpopulations at peaks of lower values. Also, they found that sharing did a better job than crowding of allocating individuals to the peaks. (12) The solution to the protein folding problem is obtained by manipulating an energy function (see Figure 30, Chapter V) full of uneven peaks and valleys. So, it is best to pursue a phenotypic sharing scheme.

The first step in phenotypic sharing is to calculate the distance (d_{ij}) between the strings in the decoded parameter space. So, for the individuals $x_i = [x_{1,i}, x_{2,i}, \ldots, x_{p,i}]$ and $x_j = [x_{1,j}, x_{2,j}, \ldots, x_{p,j}]$:

$$d_{ij} = \sqrt{\sum_{k=1}^{p} \left(X_{k,i} - X_{k,j} \right)^2}$$

Figure 16: Phenotypic distance calculation (12)

Next, let each niche be enclosed in a p-dimensional hypersphere of radius σ_{share} such that each sphere makes up 1/(number of peaks in the space) of the volume of the space. The radius of a hypersphere containing the entire space is:

$$r = \frac{1}{2} \sqrt{\sum_{k=1}^{p} \left(X_{k,\max} - X_{k,\min} \right)^2}$$

Figure 17: Radius of the hypersphere (12)

Now, we can calculate σ_{share}. If we let q equal the number of peaks in the space then:

$$\sigma_{share} = \frac{r}{\sqrt[p]{q}}$$

Figure 18: Sigma share calculation (12)

Finally, we sum a sharing function ($Sh(d_{ij})$) over all strings (see Figure 25) to get a niche count. We divide each population member's fitness by its respective niche count to complete the sharing step. Next, we allow the genetic operators to manipulate the solution strings of the present generation. Then, at the next generation, we start the niche calculations again. (12, 28)

$$Sh(d_{ij}) = 0, \text{ if } d_{ij} \geq \sigma_{share}, \text{ otherwise:}$$

$$Sh(d_{ij}) = 1 - \frac{d_{ij}}{\sigma_{share}}^{\alpha}$$

$$niche\_count = \sum_{j=1}^{N} Sh\left(d_{ij}\left(x_i, x_j\right)\right)$$

Figure 19: Calculation of niche count (12, 28)

Niching Summary

Niching offers potential benefits towards solving the protein folding problem. Because, the solution space contains many hills and valleys, niching could work to force the genetic algorithm to explore more of the space. Niching also could work when combined with conjugate gradient minimization. Another possible benefit of niching could be in combating deception. Because niching operates by de-emphasizing areas, it could move the genetic algorithm away from deceptive minima.

Summary

There are a number of techniques that can be used to enhance a genetic algorithm. We can use conjugate gradient methods to calculate local minima and then replace some, none, or all the population members. We can apply sharing to diversify the population. We can apply a combination of sharing to diverge the population and then a local minimization step to move the members closer to extrema at their various locations in the space. There are a large number of possible experiments in these areas.

V. Design, Modification, and Implementation

The purpose of this chapter is to lay out the initial AFIT implementation and then to discuss additions/modifications. Then, this chapter is to summarize the plans for experimentation.

The AFIT Implementation

The present AFIT implementation (see Figure 26) is coded in C and is located in the ~genetic directory in the Parallel Lab's account (Room 243, Bldg. 640, WPAFB).

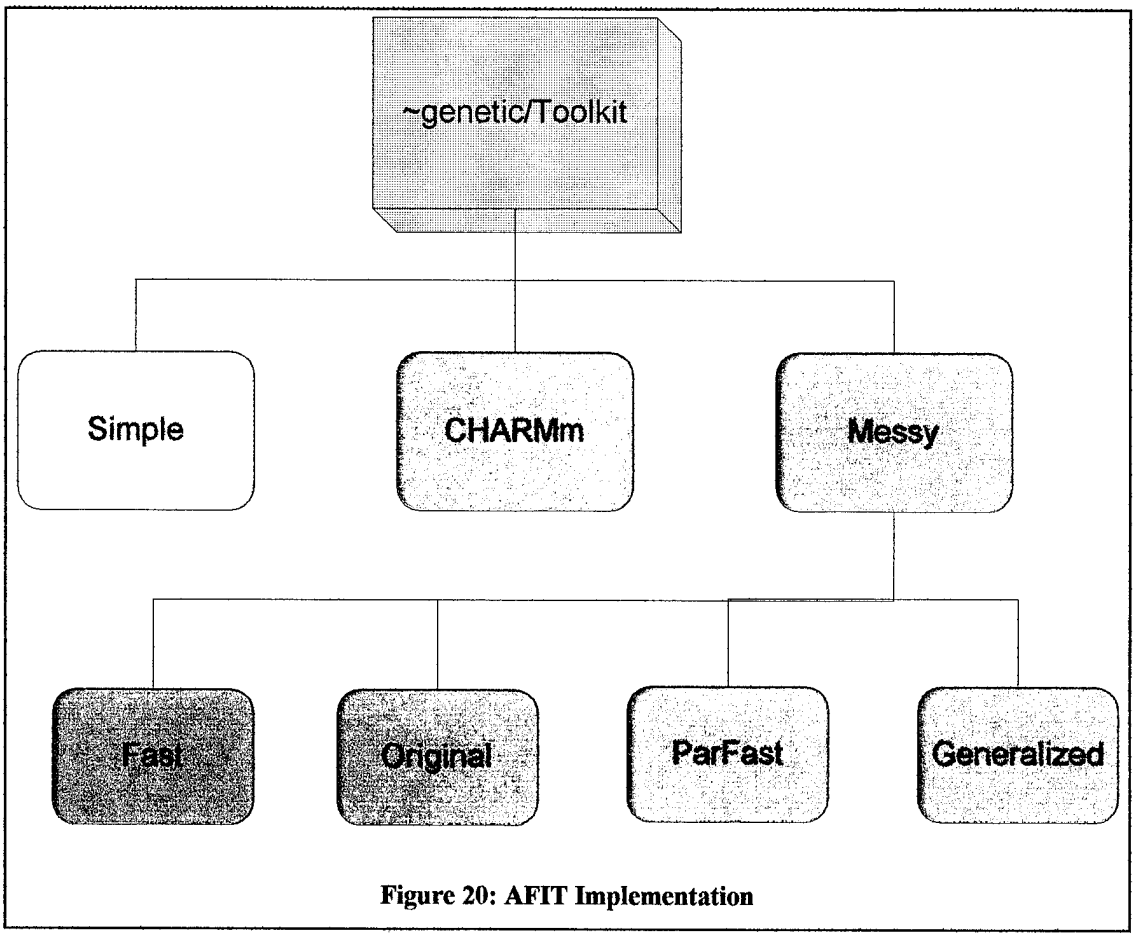


Figure 20: AFIT Implementation

There are three principal divisions of the Toolkit - *Simple*, *CHARMm*, and *Messy*. The *Simple* directory contains the code necessary for the simple genetic algorithm. It includes selection, crossover, mutation, and supporting code. The *Messy* directory is broken into subdirectories. Each of them contain specialized messy genetic algorithm code as indicated by their titles. The *CHARMm* directory contains the code necessary for the matrix representation of each conformation and for the calculation of the energy of each conformation. This code is used by **both** the simple and messy genetic algorithm implementations. The *CHARMm* directory also contains the code for encoding and decoding the population individuals into their respective dihedral angles. For example, [Met]-enkephalin consists of twenty-four dihedrals each of which are encoded by ten binary bits. So, we have a total of 240 bit solution strings that we manipulate with genetic operators. (36)

Inputs to the Implementation

One input file used by the AFIT implementation (see Figure 29) is generated by a package called Cerius2. Cerius2 produces a sequential listing of all atoms present in the molecule. This listing is called a Z-matrix (see Figure 21). The bond length is the distance between the present atom and $atom_j$. The bond angle is formed between the present atom,

[atom type] **[bond length]** [flag] **[bond angle]** [flag] **[dihedral]** [flag] [$atom_j$] [$atom_k$] [$atom_l$] [charge]

Figure 21: Z-matrix Format

$atom_j$, and $atom_k$. The dihedral is the torsion angle of the middle bond formed between the present atom, $atom_j$, $atom_k$, and $atom_l$. For example, in Figure 22, this line shows

```
C   1.49994   0   111.60606   0   -119.97103   1   3   2   1   0.000
```

Figure 22: Example line of Z-matrix

that we have a carbon atom (C). It is 1.49994 angstroms (an angstrom equals 10^{-10} meters) distant from the previous atom in the list. There is an 111.60606 degree angle (with a vertex at atom 3) that is formed between the present atom, atom 3, and atom 2. Also, there is a -119.97103 degree dihedral angle formed on the bond between atom 3 and atom 2 with respect to the chain of atoms extending from the present atom to atom 1. The flags set to 0 indicate that parameters are held fixed. The charge field of the Z-matrix is not used. A separate file, called the residue topology file (RTF), is produced by a package called QUANTA. This file contains data about atomic charges and specific atom type information. Also supplied as an input to the implementation is a parameter file (PARM) that is also produced by QUANTA. This file contains constant parameters associated with bond lengths, bond angles, dihedral angles, and non-bonded pairs. Lastly, there is a user-supplied file, called *in*, that contains run-time parameters such as population-size, number of experiments, and other options (see Figure 48). (5, 6)

Outputs of the Implementation

The output of the AFIT implementation (see Figure 29) comes in several forms. First, there is a file generated (or appended to if it already exists) called *out*. It contains a line for every generation containing data such as the number of trials, percent converged, minimum energy at that generation, and the average energy of that generation. Also, the implementation writes to a file called the PDB file. The PDB file contains Cartesian

coordinates corresponding to each atom. The Cartesian coordinates are important for the evaluation of the energy function of a particular conformation. Finally, there is user-generated output (by *printf* statements) which defaults to the screen or can be directed to a file. (5, 8, 36)

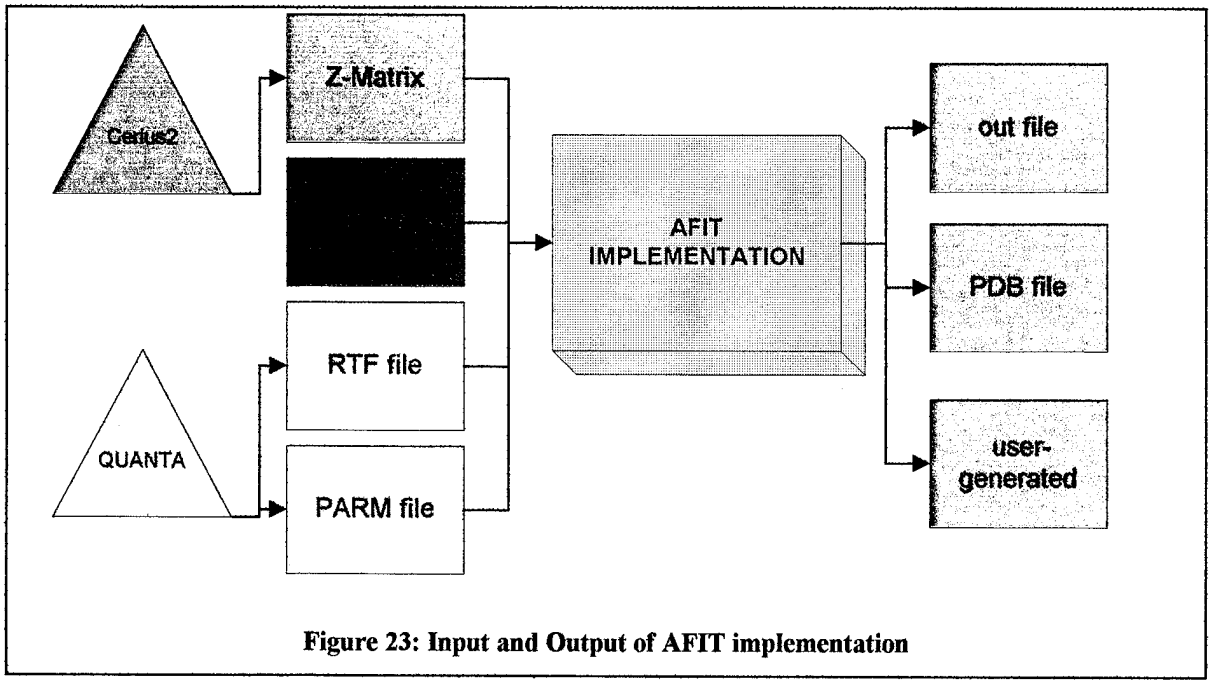


Figure 23: Input and Output of AFIT implementation

Modifications/Additions to Implementation

This section details the modifications and additions to the AFIT implementation. The topics to be discussed are local minimization (derivation and techniques), niching, and the use of tournament selection (fitness disproportionate selection) in a simple genetic algorithm.

Local Minimization

One of the principal objectives of this experimentation is to determine what method of applying local minimization (if any) works best with regards to effectiveness and efficiency when used as a possible enhancement of the AFIT implementation. In exploring different techniques of local minimization, we settled on using a conjugate gradient technique discussed by Press *et al* (51) primarily because the code was readily available and looked relatively simple to adapt for use with the AFIT implementation. This minimization technique required the use of the first derivative of the function being minimized. Thompson (56) discusses a promising method of calculating first derivatives but the AFIT implementation's representation of the molecule had to be altered to allow for Thompson's method of calculating of the first derivative of the energy function (see Figure 24). So, before local minimization routines could be inserted into the code, a number of steps (which are detailed in this section) were necessary:

(1) AFIT implementation was altered to allow for Thompson's representation of the molecule;

(2) Thompson's Cartesian coordinate system transformation was implemented;

(3) Thompson's derivative method was used to calculate the partial derivative representing the change in position with respect to the change of the dihedral;

(4) Partial derivative representing the change in the distance with respect to the change in position was calculated;

(5) Partial derivative representing the change in the energy with respect to the change in interatomic distance was computed;

(6) Partials were multiplied together resulting the first derivative which represents the non-bonded energy with respect to a particular dihedral angle;

(6) Derivatives with respect to all dihedral angles were used in conjugate gradient routine to minimize energy function. (42, 51, 56)

$$E = \sum_{(i,j)\in B} (K_{rij}(r_{ij} - r_{eq})^2) + \sum_{(i,j,k)\in A} (K_{\Theta ijk}(\Theta_{ijk} - \Theta_{eq})^2) + \sum_{(i,j,k,l)\in D} (K_{\Phi ijkl}(1 + \cos(n_{ijkl}\Phi_{ijkl} - \gamma_{ijkl}))) + \sum_{(i,j)\in N} ((A_{ij}/r_{ij})^{12} - (B_{ij}/r_{ij})^6 + q_i q_j/4\pi\varepsilon r_{ij})$$

where: K_{rij}, $K_{\Theta ijk}$, $K_{\Phi ijkl}$, r_{eq}, Θ_{eq}, n_{ijkl}, γ_{ijkl}, A_{ij}, and B_{ij} are empirical constants
B - bonded atoms, A - atoms forming bond angles, D - atoms forming dihedral angles,
N - non-bonded atoms (atoms with more than 3 bonds separating them)
r_{ij} - bonded (or non-bonded) atom term, Θ_{ijk} - bond angle term, Φ_{ijkl} - dihedral term

Figure 24: CHARMm energy model (38)

Cartesian Coordinate Transformation of Implementation

Thompson's molecular representation is based first on each atom of the molecule being in its own coordinate system. Then, after each coordinate system is calculated, we calculate the coordinates of all atoms with respect to the same coordinate system. (56)

Background

First, suppose we have three atoms: A_1, B_2, and B_3 where B_2 lies on A_1's negative x-axis and B_3 is the next atom in the sequence (see Figure 25) . Each atom is in its own

coordinate system. The origin is at A_1 and our goal is to find the coordinates of atoms B_2 and B_3 with respect to A_1's coordinate system.

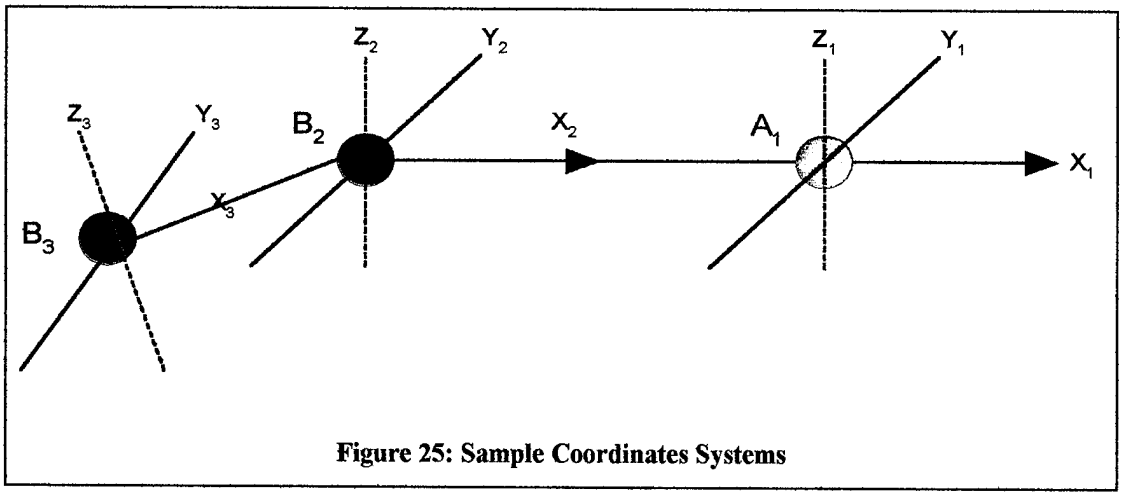


Figure 25: Sample Coordinates Systems

When we use Thompson's transformation to put B_2 and B_3 in terms of A_1's coordinate system, they are then referred to as A_2 and A_3 respectively. First, we represent each atom with a four by four matrix (see Figure 26). The first three numbers of its first column is a unit vector along its x-axis, the first three numbers of its second column is a unit vector along its y-axis, and the first three numbers of its third column is a unit vector along its z-axis. Finally, the first three numbers of the fourth *column* are the atom's actual coordinates. The fourth *row* is 0 0 0 1 for computational purposes. (42, 56)

$$\begin{matrix} -\cos\alpha & -\sin\alpha & 0 & -R\cos\alpha \\ \sin\alpha\cos\beta & -\cos\alpha\cos\beta & -\sin\beta & R\sin\alpha\cos\beta \\ \sin\alpha\sin\beta & -\cos\alpha\sin\beta & \cos\beta & R\sin\alpha\sin\beta \\ 0 & 0 & 0 & 1 \end{matrix}$$

Where α is the bond angle, β is the dihedral angle, and R is the bond length

Figure 26: B-Matrix representation of an atom (56)

For the first three atoms of the protein, there are no bond (with only two atoms) or dihedral (with only three atoms) angles. So, by definition, β_2 (dihedral value for atom 2's B Matrix) is set to π while β_3 (dihedral value for atom 3's B Matrix) and α_2 (bond angle value for atom 2's B Matrix) are set to 0. Figure 27 and Figure 28 show the resulting B matrices that result from these defined bond and dihedral angle values.

$$B_2 = \begin{bmatrix} -1 & 0 & 0 & -R_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 27: B_2 Matrix

$$B_3 = \begin{bmatrix} -\cos\alpha_3 & -\sin\alpha_3 & 0 & -R\cos\alpha_3 \\ \sin\alpha_3 & -\cos\alpha_3 & 0 & R\sin\alpha_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 28: B_3 Matrix

Also, by definition, A_1 is just a four by four identity matrix. Note that its coordinates (first three numbers in the fourth column) are 0,0,0.

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 29: A_1 Matrix

To find A_2, (B_2 in terms of A_1's coordinate system), we multiply the A_1 matrix by the B_2 matrix ($A_2 = A_1 \mathbf{X} B_2$). Because A_1 is an identity matrix, $A_2 = B_2$.

$$A_2 = \begin{bmatrix} -1 & 0 & 0 & -R_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 30: A_2 Matrix

To find A_3, (B_3 in terms of A_1's coordinate system), we multiply the A_2 matrix by the B_3 matrix ($A_3 = A_2 \mathbf{X} B_3$).

$$A_3 = \begin{bmatrix} \cos\alpha_3 & \sin\alpha_3 & 0 & R_3\cos\alpha_3 - R_2 \\ \sin\alpha_3 & -\cos\alpha_3 & 0 & R\sin\alpha_3 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 31: A3 Matrix

From this point on, for every atom we wish to add to the structure, we first calculate its B matrix. Then, we multiply its B matrix by the A matrix of the adjacent atom to get the A matrix of the newly added atom ($A_{new} = A_{adj} \mathbf{X} B_{new}$). We continue these steps until all atoms are in the base coordinate system. (42, 56)

Implementation of Cartesian Coordinates System

The primary change to the existing implementation was the addition of the A and B matrix structures. The dihedral and bond angles are already stored and easy to access in calculating the matrices. After assigning initial values, all that was really required was an insertion of a loop that would step through each atom and calculate and store its B matrix. Then, a simple cross product function is called to calculate each A matrix. The only stumbling block was what became termed the *atom 42 problem*. As stated earlier, [Met]-enkephalin is our primary testing molecule. Observing Figure 1, we see that atom 42 is added adjacent to atom 2. Therefore, we can not compute its matrices by multiplying other matrices (as described in the above section). So, it was necessary to hard-code the values of the matrix for that atom. This becomes an important consideration when using the AFIT implementation for *other molecules*. While the implementation is generic enough to run different molecules by using different input files, when computing the matrix system for a molecule, any of its atoms added adjacent to its first two atoms must have their values hard-coded.

Calculation of first derivative

It was now necessary to calculate the derivative of the non-bonded energy with respect to the dihedral angle:

$$\frac{\partial E_{nb}}{\partial \beta_x} = \frac{\partial E_{nb}}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial q_i} \frac{\partial q_i}{\partial \beta_x}$$

Figure 32: Derivative of the non-bonded energy

First, Thompson provides a formula for calculating the change in position with respect to the change of the dihedral:

$$\frac{\partial \mathbf{q}_i}{\partial \beta_x} = a_{x,k'} \times [q(j) - q(k')]$$

Figure 33: Positional Partial Derivative (56)

where $a_{x,k'}$ is a unit vector along the X-axis of a *chain* atom, q(j) is the position vector in the base coordinate system of the atom we are adding, and where q(k') is the position vector in the base coordinate system of a *chain* atom. A **chain atom** is on the atom chain between the atom we are adding and the base atom (A_1). All that was needed was to add a loop that took the difference of the two vectors and then called the cross product function for each atom. (42, 56)

$$\frac{\partial \mathbf{r}_{ij}}{\partial \mathbf{q}_i} = \frac{\partial r_{ij}}{\partial x_i}, \frac{\partial r_{ij}}{\partial y_i}, \frac{\partial r_{ij}}{\partial z_i}$$

$r^2 = x^2+y^2+z^2 \rightarrow 2r\partial r/\partial x = 2\Delta x \rightarrow \partial r/\partial x = \Delta x/r$

(Δy and Δz terms of the ∂r vector are derived the same way)

where:

$\Delta x = x_i - x_j$ $\Delta y = y_i - y_j$ $\Delta z = z_i - z_j$

Figure 34: Calculation of interatomic distance partial

Next, we can compute the change in the distance with respect to the change in position (see Figure 34) which can be calculated from Thompson's arrangement using *atom i* (atom we are adding) and *atom j* (atom adjacent to the one we are adding) using the coordinate (fourth) column of their respective A matrices. This was accomplished in the code by looping through the atoms and performing subtraction and division steps to calculate the vector.

Finally, deriving the formula for the change in the energy with respect to the change in interatomic distance had to be accomplished by hand (there was no given formula). Recall that the non-bonded term of the energy function is:

$$\sum_{(i,j)\in N} ((A_{ij}/r_{ij})^{12} - (B_{ij}/r_{ij})^{6} + q_i q_j/4\pi\varepsilon r_{ij})$$

Figure 35: Non-bonded term of the energy function

So, taking its derivative with respect to the interatomic distance (r_{ij} term) we get:

$$\sum_{(i,j)\in N} (-12(A_{ij})^{12}(r_{ij})^{-13} + 6(B_{ij})^{6}(r_{ij})^{-7} - q_i q_j/[4\pi\varepsilon(r_{ij})^{2}])$$

Figure 36: Derivative of the non-bonded term

Now it was just necessary to set up a loop in the code that inserted the proper values into the formula to generate a scalar that is the partial derivative of the non-bonded term with respect to the change in the interatomic distance. (42)

Finally, all that remained to do was to multiply the ∂r vector by the above scalar and then perform a simple dot product of the resulting vector with the ∂q_i vector. The resulting *scalar* is the derivative of the non-bonded energy with respect to a **particular** dihedral angle. So, this procedure must be repeated for *every dihedral* resulting in an array of derivatives. (42, 56)

Fletcher-Reeves-Polak-Ribiere Algorithm

Press, *et al*, (51) discuss a conjugate gradient (see discussion in Chapter IV and Chapter V) algorithm/implementation that is a combination of the Fletcher-Reeves and Polak-Ribiere optimization methods. This technique uses the derivatives (which we calculated previously) of the function being optimized.

Discussion

The Fletcher-Reeves and Polak-Ribiere methods are nearly identical. They are based on the calculation of a sequence of *mutually orthogonal* vectors ($\mathbf{g_x}$) and the calculation of a sequence of *mutually conjugate* vectors ($\mathbf{h_x}$). So, symbolically, $g_i \bullet g_j = 0$ and $h_i \bullet A \bullet h_j = 0$ (where A is a symmetric *nxn* matrix). (51)

In calculating the sequences of vectors, two sequences of constants (γ_x, λ_x) are used such that $g_{i+1} = g_j - \lambda_i A \bullet h_j$ and $h_{i+1} = g_{i+1} + \gamma_i h_j$ where $\lambda_i = (g_i \bullet g_i)/(g_i \bullet A \bullet h_i)$ and $\gamma_i = -(g_{i+1} \bullet A \bullet h_i)/(h_i \bullet A \bullet h_i)$. It can be shown that:

$$\gamma_i = (g_{i+1} \bullet g_{i+1})/(g_i \bullet g_i) = ((g_{i+1} - g_i) \bullet g_{i+1})/(g_i \bullet g_i)$$

Figure 37: γ_i equation (51)

It is at this point that the difference between the two techniques comes out. The Fletcher-Reeves approach sets $\gamma_i = ((g_{i+1} \bullet g_{i+1})/(g_i \bullet g_i))$ (from Figure 43) while the Polak-Ribiere approach sets $\gamma_i = ((g_{i+1} - g_i) \bullet g_{i+1})/(g_i \bullet g_i)$ (from Figure 43). These two values for γ_i are equal only for exact quadratic forms. So, the Polak-Ribiere provides for proceeding beyond the minimums of the quadratic forms to possibly lower minimums. By changing a line of the implementation, the code can be switched between the two methods. The mutually conjugate and mutually orthogonal vectors are used to step toward a local minimum. (51)

Implementation

One of the primary motivating factors of choosing this local minimization technique was the fact that an implementation (with some documentation!) was readily

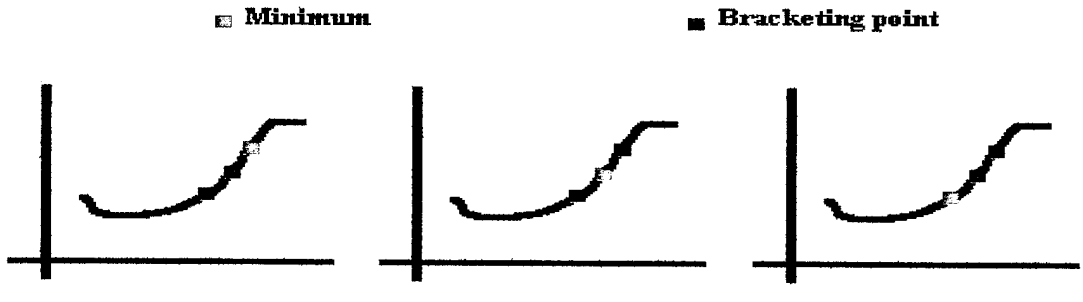


Figure 38: Examples of bracketing a point

available. After minor modifications, `frprmn.c` and its supporting functions were placed into the AFIT implementation. Gates (18) created a version of the bracketing function used in place of the function `mnbrak.c`. Gates' version, called `mymnbrak.c`, brackets the current minimum with two other points. The code checks and resets those points if necessary to make sure the point is indeed bracketed. Once the point is properly bracketed, a new middle point is found between the lower bracketing point and the original middle [steps 2-4 below]. That new middle becomes the upper bracket point and the old lower bracket point becomes the next middle. Then a new low bracket point is found. Now, the cycle starts over again with us finding a new middle between the existing middle and the lower bracket point [steps 5-6 below]. This cycle continues until the

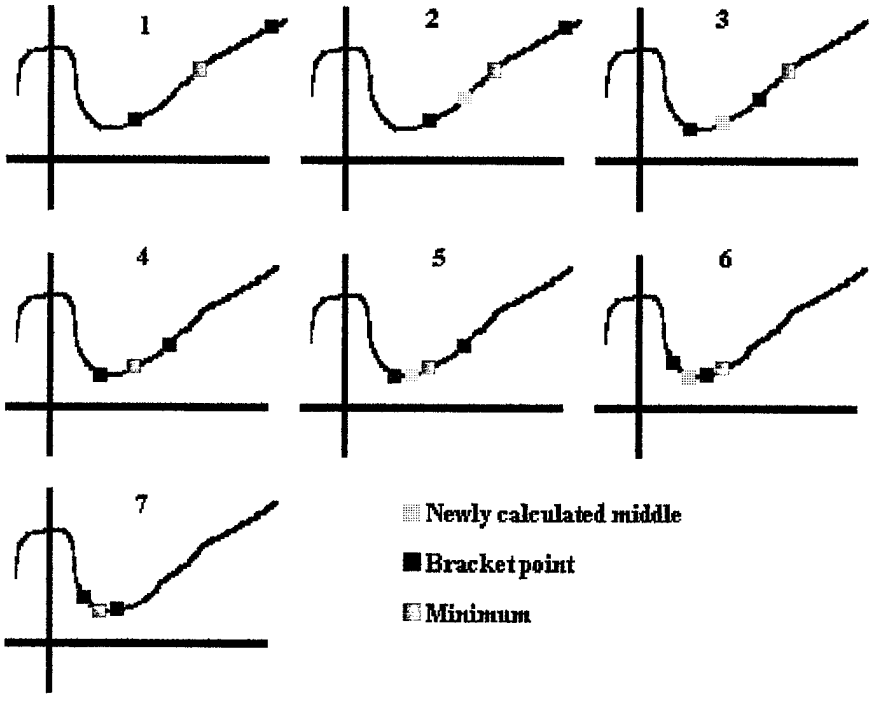


Figure 39: Local minimization bracketing steps (42)

middle point is lower than both the bracketing points where it is assumed that we have reached the local minimum [step 7]. (42, 51)

Implementation of Niching

The niching implementation follows from the niching algorithm steps (see Figures 17-20) presented in Chapter IV. As stated in Chapter IV, phenotypic sharing was chosen because the function we are dealing with is filled with uneven peaks and Goldberg (12, 28) discouraged the use of genotypic sharing and crowding for such functions. A general algorithm for phenotypic sharing is as follows:

1) Calculate Distances
2) Calculate Hypersphere
3) Calculate σ_{share}
4) Calculate Sh(d)
5) Calculate niche count
6) Divide fitness by niche count

Figure 40: Phenotypic Sharing Algorithm

In calculating the distance, we are dealing with the string components which are dihedral angles. So, we convert the set of strings into a two dimensional array of 24 dihedrals for each population member. This decoding of the strings is accomplished by the using the mapping $D:\{0,1\}^{10} \rightarrow [-\pi,\pi]$ of ten bit subsequences to dihedral angles such that:

$$D(a_1, a_2, \ldots, a_{10}) = -\pi + 2\pi \sum_{j=1}^{10} a_j 2^{-j}$$

Figure 41: Dihedral decoding scheme (45)

This encoding gives us a precision of approximately one-third of a degree. Now, we are subtracting angles that fall in the -Π to Π radians range. However, we can not do just a straight subtraction as we want to keep the result in that range as well. So, we must take an absolute value of the difference and then subtract that from two PI. Then, we take the *minimum* of that initial difference or the value obtained by subtracting it from two PI. For

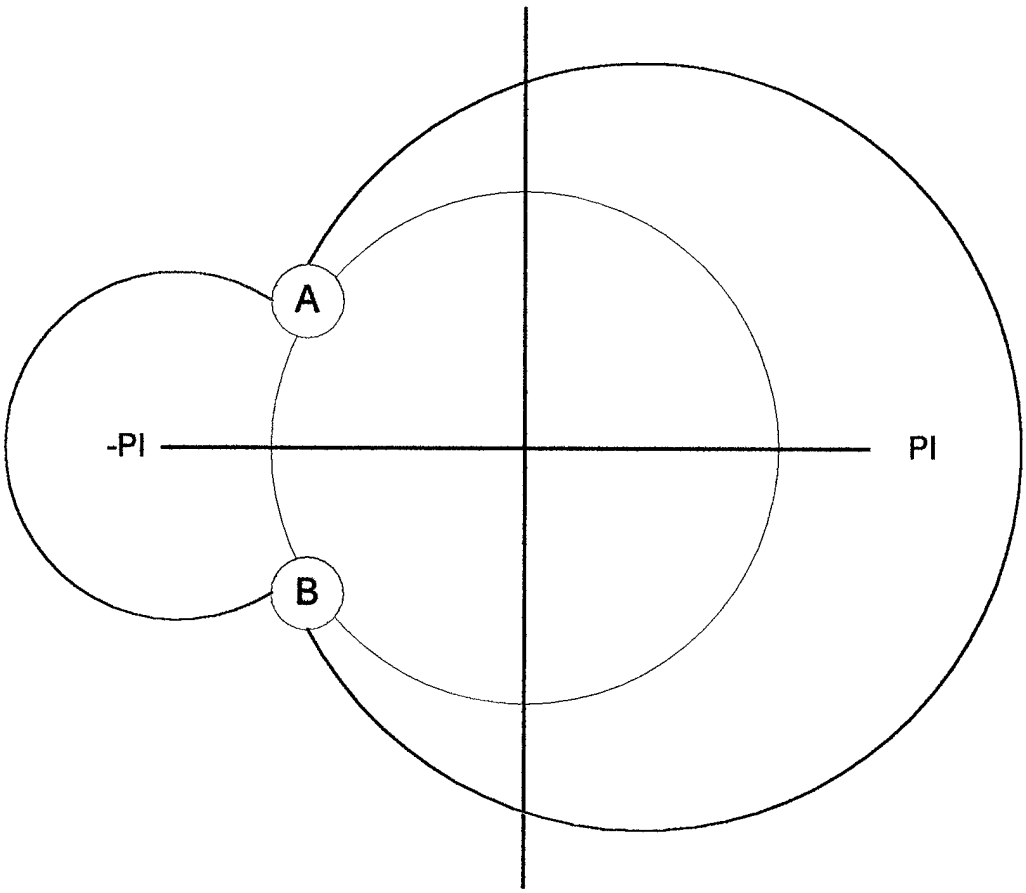


Figure 42: Distance Calculation of Dihedral Angles

example, if we are finding the distance between angles A and B in Figure 42, we would get the shorter, red distance instead of the longer, blue distance. Then, to finish the distance calculation, we square the distance we calculated and continue on with the next

two angles and keep summing the squares of their distances. Finally, we take the square root of the total squares of the dihedral distance for each population member. (45)

Next, to calculate the radius of the hypersphere, we do not have to use the formulas provided by Deb and Goldberg (12). Instead, we know by the nature of our space that the complete area covered is the square root of the number of dihedrals times PI. So, this step is accomplished by a simple multiplication. Then, the calculation of σ_{share} is accomplished by dividing the radius by the value obtained when you take the number of desired peaks to the one over number of dihedrals power. Finally, we use Deb and Goldberg's (12) formula for Sh(d) and accumulate those values for each member of the population. Those accumulated values are divided into the respective individual's fitness to accomplish the appropriate de-emphasis of fitness. This code is contained in the file `niche.c` which is located in the `~genetic/Toolkit/Simple` directory. (36)

Implementation of Tournament Selection in the Simple GA

Another modification/addition to the AFIT implementation was putting the tournament selection option into the simple genetic algorithm. Code for tournament selection already existed for the messy genetic algorithm implementations. There is a slight difference in the way the messy genetic implementation represents the population compared with that of the simple genetic implementation. Both use record types, but the messy genetic implementation has more elements defined in its population record. So, the code had to be altered slightly to allow for handling a different record configuration.

After that, all that remained was to copy the code over into the *Simple* directory and make some other minor variable name changes.

Genetic Algorithm Implementation Options

Figure 43 shows an example of the user's *in* file. Note the *Options* line which is third from the bottom. This line allows the user to set the implementation to run a certain way. The options (See Figure 44 for a partial listing of options now available) shown in

```
      Experiments = 1
     Total Trials = 10000
  Population Size = 20
 Structure Length = 240
   Crossover Rate = 0.65
    Mutation Rate = 0.003
   Generation Gap = 1.0
   Scaling Window = 1
  Report Interval = 1
 Structures Saved = 1
Max Gens w/o Eval = 10
    Dump Interval = 0
      Dumps Saved = 0
          Options = nye
  Number of Peaks = 16777216.0
      Random Seed = 987654321
```

Figure 43: Example "in" file

the example would have the AFIT implementation do niching, use fitness proportionate (roulette wheel) selection, and use elitism. Note that the number of peaks is set to 16777216 which equals 2^{24}. In other words, we treat the entire hypersphere as if it were divided into 2^{24} areas for the solutions to cluster in. Note that in order to force the

implementation to replace a percentage of strings (or components) other than zero or one-hundred, the "Z" option must be used *and the source code must be modified*. See Appendix C for instructions on altering the replacement percentage.

option	flag that is set	description
'E':	Lamarckflag	use 100 percent replacement after local min
'F':	Fivepercentflag	do a local min every 20th generation
'm':	Minimizationflag	locally minimize
'n':	PShareflag	use phenotypic sharing in niching
'p':	MutateMinflag	when a mutation occurs, do local min
'T':	TenLMflag	Start locally minimizing after ten generations
'y':	FitProflag	use fitness proportionate selection
'Y':	TSflag	use Tournament Selection
'z':	EndLMflag	do local minimization at last generation
'Z':	Davisflag	replace only a percentage of the strings

Figure 44: Partial listing of "in" file Options

Summary

This chapter has discussed the current AFIT implementation as well as its inputs and outputs. Then, this chapter detailed the additions/modifications (see Appendix D for a listing of those additions/modifications) that have been made to the AFIT implementation including local minimization, niching, and tournament selection (for the simple genetic algorithm). Some of the areas covered in the addition/modification section are techniques of implementation and the motivations behind some of the design decisions. The chapter then concluded with a brief discussion on the *in* file options.

VI. Experimentation and Analysis

The purpose of this chapter is to define experimental design, detail the experiments, and analyze their results. These experiments attempt to find better ways of obtaining quality solutions (structures) to the protein folding problem. Experiments are important because while they *prove* nothing, they can be used to observe tendencies. We can conduct experiments on a set of data to learn the *nature* of that data. This chapter discusses motivations, expectations, and results of the experiments. This chapter concludes with a comparison of various strategies and combinations of those strategies.

Design of Experiments

As stated earlier, the protein molecule model on which the experiments are based is [Met]-enkephalin. The minimum energy value (from now on referred to as the *optimum* solution) found in QUANTA for this protein is -29.225 kcal/mol (17). The experiments (with a combined total of over 8000 CPU-hours of execution time) focus on trying to approach that value which puts us closer to having the "correct" folded structure dihedral angles (see Figure 45).

Residue	ϕ	ψ	ω	χ_1	χ_2	χ_3	χ_4
Tyr:	-86.13	156.0	-176.84	-172.62	78.75	165.94	
Gly:	-154.34	83.67	168.75				
Gly:	83.67	-73.83	-170.16				
Phe:	-137.11	19.34	-174.02	58.71	-85.43		
Met:	-163.48	160.31	-179.65	52.73	175.08	-180.00	-58.36

Figure 45: Dihedral angles (in degrees) for accepted *optimum* of [Met]-enkephalin(17)

The goal of these experiments is (with a high level of confidence) to determine which genetic algorithm strategy or combination of genetic algorithm strategies offers the best or better chance of achieving a minimum energy conformation. The experiments are organized as follows:

- For Conjugate Gradient local minimization test and compare dihedral and string replacement percentages using:
  - Roulette Wheel (Fitness Proportionate) Selection
  - Tournament (Fitness Disproportionate) Selection
- For Niching test for:
  - Performance when varying number of peaks
  - Performance when combined with delayed replacement strategies

In the execution of these experiments, attention is to be focused on the following quantitative and statistical comparisons which serve as an indicator of solution quality:

- Lowest average energy
- Lowest minimum energy
- Execution times

In several cases, the average energies of the experiments are similar. When Wilk-Shapiro normality tests were performed on the populations of energies, the average energies were shown to be from populations that were *not* normally distributed. Kruskal-Wallis tests were then used to determine if significant differences existed between the averages.

When viewing the graphs (for instance, see the graph of Figure 46), note that frequently the graphs are plotted starting with generation five, ten, twenty-five, or fifty rather than starting with generation one. This is because generation one energy data is

often quite high such that it causes the graph to be spread over too high of a range of energies. This results in some difficulty in viewing the graph in later generations where the data values are closer together. So, by starting with a later generation (we are not real concerned, of course, with early generations), it becomes easier to see differences in the effectiveness of various strategies.

Local Minimization Experiments

For the first series of experiments, we are interested in knowing what percentage of replacement would work best for our application. Davis and Orvosh (49) report that for their applications a replacement percentage of five percent worked the best. In other words, local minimization is performed on all solution strings and then five percent are arbitrarily replaced. There are two approaches to replacement that were tried. The first approach was to replace a percentage of the components of each string. In other words, for *five percent replacement*, five percent of the dihedrals in each string were arbitrarily replaced. The second approach was to use replacement as discussed by Davis and Orvosh (49) which was to replace a percentage of the strings.

Replacement of components

First, a series of experiments used a simple genetic algorithm with fitness proportionate selection on a population of fifty individuals for 6000 trials. A population size of fifty was selected for the experiments in an attempt to remain consistent with previous research (17). For each set of the experiments, the random seeds 987654321,

989954321, 998954321, 999854321, and 99954321 were used (in fact, these random seeds were used for all experiments in this research for uniformity). The percentage of replacement (of the dihedral angles) was varied from five to fifteen. Also, these experiments are viewed with a set of runs with zero percent replacement (Baldwinian - see Chapter IV) and with one-hundred percent replacement (Lamarckian - see Chapter IV).

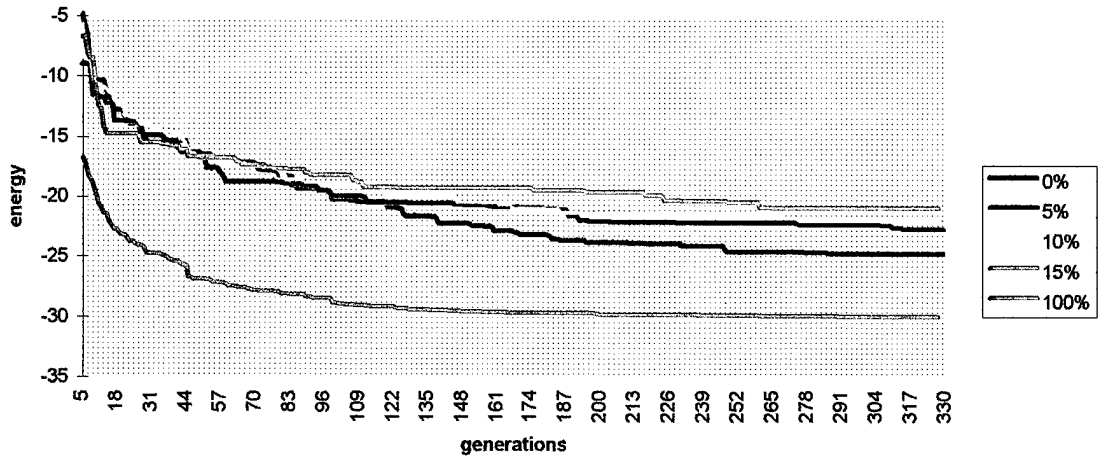


Figure 46: Fitness Proportionate comparison of dihedral replacement percentages

The graph of Figure 46 is based on five-run averages of the minimum energy found at each generation. Observe that the five, ten, and fifteen percent replacement strategies all achieve higher average minimum energies than the strategies of replacing the entire strings or not replacing anything. Moreover, in these experiments, the ranking of the dihedral replacement strategies seems directly related to the amount of dihedrals replaced. Fifteen percent had the highest average minimum energy (-21 kcal/mol), followed by ten percent (-22.2 kcal/mol), and then five percent (-22.8 kcal/mol). For this particular set of runs, the

one-hundred percent replacement strategy reaches a much lower average minimum energy (-30 kcal/mol) compared to the other techniques. While, it does not *prove* anything, this experiment does indicate that with fitness proportionate selection, we are probably better off replacing everything than just a few dihedral angles following local minimization.

Next, experiments were run to look at the possible benefits of using tournament selection with the simple genetic algorithm. In addition, there were tests for possible benefits of using dihedral replacement strategies with tournament selection. The tournament selection experiments are run in sets of five using a population size of fifty.

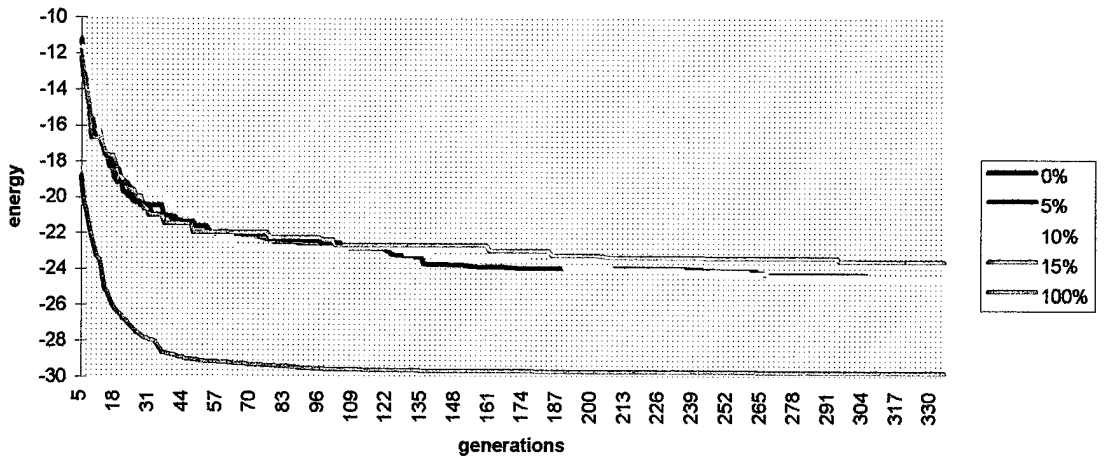


Figure 47: Tournament selection dihedral replacement experiments

For comparison purposes, a graph is plotted based on the average minimum energy found by the experiments at each generation. In Figure 47, we see results very similar to those found by the fitness proportionate replacement experiments. Once again, the one-hundred percent replacement has reached a much lower average minimum energy (around -29

kcal/mol) as compared to the other methods. The other methods have found average minimum energies at around -22 to -24 kcal/mol. So, the tournament selection dihedral replacement strategies appear to be more effective than their fitness proportionate counterparts. However, the results indicate that once again we would probably be better off replacing everything than just replacing a few dihedral angles at each generation.

Summary of component replacement experiments

Notice that for both the fitness proportionate (see Figure 46) and tournament (see Figure 47) selection strategies that the replacement of the entire strings achieved lower average minimum energies. One conjecture for this behavior may be that by replacing only a percentage of the dihedrals, we are omitting "good" dihedrals and keeping "less good" dihedrals. Thus, we are possibly inhibiting (rather than helping) the progress of the simple genetic algorithm. On the other hand, replacing the entire string forces the implementation to keep all the dihedrals which improves on the progress of the simple genetic algorithm. The poor solution quality of the dihedral replacement strategies is an indicator that for this application we should experiment with replacing the entire strings.

Replacement of strings

The next set of experiments examines the concept of arbitrarily replacing a percentage of the population members (entire strings) at each generation of the simple genetic algorithm. For these experiments, a population size of fifty was used over 12000 trials. The graphs show the results of five-run average minimum energies at each

generation. The string replacement strategies are tested using both fitness proportionate and tournament selection.

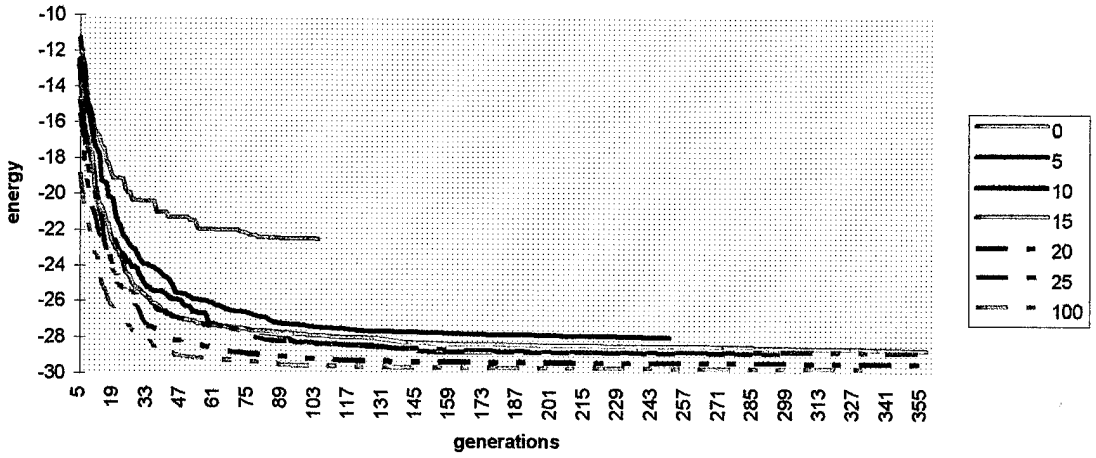


Figure 48: Tournament selection string replacement strategies

Figure 48 shows a comparison of string replacement strategies on a simple genetic algorithm that uses tournament selection. First, notice that the zero percent replacement (Baldwinian strategy) converges quickly to a higher energy value than the other replacement percentages. Why do the Baldwinian experiments converge so quickly? One conjecture is that the combination of the aggressive tournament selection and the non-replacement of strings causes members of the population with poor fitnesses to quickly be excluded from the population. This results in applying the genetic operators to similar strings which causes rapid convergence to relatively poor solutions. Notice also that the five percent replacement strategy converges (though not as early as the Baldwinian approach) at a slightly higher average energy (around -27 kcal/mol). The Lamarckian strategy, on the other hand, has the best average energy at almost -30 kcal/mol. In other

words, it is *on average* finding conformations with a lower energy than the *optimum* conformation. Note that the other replacement percentages of ten, fifteen, twenty, and twenty-five all reach average energies below -28 kcal/mol which shows that they are all averaging close to the energy of the *optimum* solution. All the replacement percentages perform well with the exception of replacing zero or five percent.

	0%	5%	10%	15%	20%	25%	100%
min	-24.6118	-29.5072	-31.1748	-30.99	-32.093	-32.4905	-31.3162
ave	-22.5944	-28.254	-29.0797	-28.7621	-29.4104	-28.8546	-29.7675
dev	1.913753	1.428581	2.10214	1.312718	1.72258	2.14637	1.18114

Figure 49: Statistical comparison of Tournament Selection string replacement minimum energies

Figure 49 provides another way to analyze the tournament selection strategies. It shows the overall minimum energy found by each strategy, the average of the minimum energies found per strategy, and the standard deviation of the minimum energies found by each strategy. Note by the standard deviations, that the strategies are rather consistent in the minimum energies found. This is also apparent when comparing the averages to the overall minimum energy found by each technique. In fact, through Kruskal-Wallis tests, it has been shown that there is no significant difference between these averages. Finally, observe that the ten, fifteen, twenty, twenty-five, and one-hundred percent string replacement strategies all found a conformation with an energy less than -30 kcal/mol — they each found a conformation with an energy lower than the *optimal* solution!

In comparison (see Figure 50), the fitness proportionate selection technique employed with the various string replacement strategies also perform well. An easy observation of the graph is that the zero percent replacement (Baldwinian strategy) does not perform as well as the other strategies. However, notice that the fitness proportionate Baldwinian strategy does not converge rapidly like the tournament selection version. This is because fitness proportionate selection allows for existence of population members with poorer fitness which creates diversity. Diversity has the effect of slowing convergence.

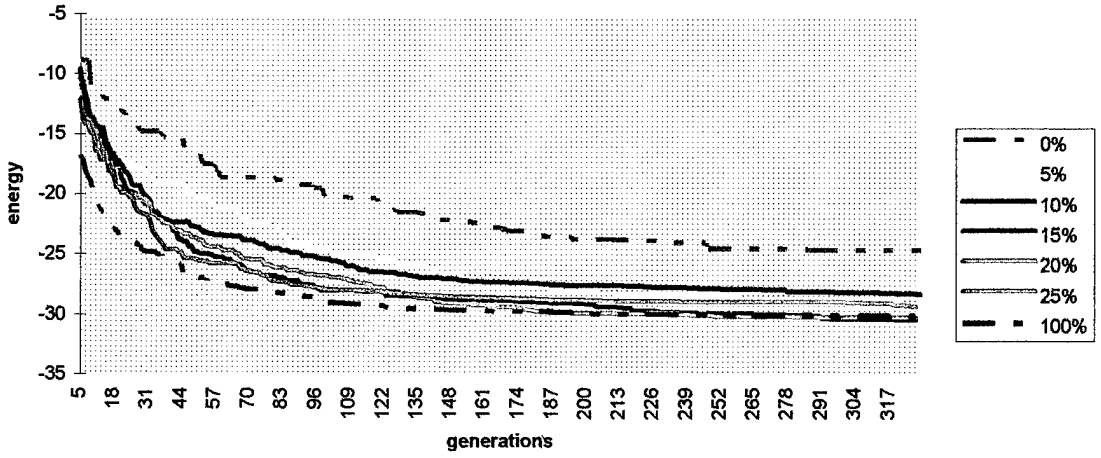


Figure 50: Fitness proportionate string replacement strategies

Note that the fifteen, twenty, and one-hundred percent replacement strategies all cluster around an *average* minimum energy of -30 kcal/mol. Figure 51 further differentiates between the replacement strategies through some simple statistical analysis of the minimum energy found by each experiment.

Figure 51 contains the absolute minimum energy, the average minimum energy, and the standard deviation of the minimum energies found by the various replacement experiments. While there was greater variation in the minimum energies found by the twenty percent replacement strategy, it found the lowest energy of all experiments, -35.1889 kcal/mol. Note that the fifteen percent replacement strategy has the best *average* minimum energy while it found only the third best minimum energy. Also, observe that every strategy except the zero percent replacement found a minimum energy of less than -31 kcal/mol. In fact, the average minimum energy found by the zero replacement experiments is at least 5 kcal/mol higher than the average minimum energy of all the other strategies.

	0%	5%	10%	15%	20%	25%	100%
min	-27.6581	-34.0205	-31.3275	-33.9411	-35.1889	-31.0853	-32.8813
ave	-24.8085	-29.1842	-28.3492	-30.6108	-30.4155	-29.4659	-30.1201
dev	1.868289	2.814428	2.115062	2.56697	3.076521	0.984139	2.377013

Figure 51: Statistical comparison of fitness proportionate replacement strategies

Another way of analyzing how close we are to finding the *optimum* solution is to compare the solution strings of a population for similarity (in terms of common bits) to the *optimum* string. The solution string in Figure 52 is the most similar string (of the population of fifty strings) from a one-hundred percent replacement experiment to the *optimum* string (see Figure 53).

```
00000010000110011000000000000110110101110011101011000000000010
00001101011001011110000000001000001100110101010111000001010 0
00001001000000000101101110010011111100111011001001000001010 1
11111100110011100010010011000001001011001001100110111111010 01
```

Figure 52: Most similar string of fitness proportionate 100% replacement experiment to *optimum*

```
01000010111111011110000000010010001001001101110111011111000 00
10111011100100101110000001110000011110101000110111000000100 01
00001011110000010101101110000010101001110100001101101001011 0
11111100100000000000111100100011110110000101011010000000000 1
```

Figure 53: Bit representation of *optimal* conformation of [Met]-enkephalin

146 bits in common
Order of the bits in common

11 order (2^9) bits in common
16 order (2^8) bits in common
15 order (2^7) bits in common
13 order (2^6) bits in common
15 order (2^5) bits in common
15 order (2^4) bits in common
15 order (2^3) bits in common
14 order (2^2) bits in common
16 order (2^1) bits in common
16 order (2^0) bits in common

Correlation Matrix

11	0	0	0	0	0	0	0	0
9	16	0	0	0	0	0	0	0
8	10	15	0	0	0	0	0	0
7	9	11	13	0	0	0	0	0
7	8	9	11	15	0	0	0	0
5	6	7	8	10	15	0	0	0
3	4	5	6	7	11	15	0	0
1	1	2	3	3	6	8	14	0
1	1	1	1	1	2	4	8	16
0	0	0	0	0	1	3	6	12

Figure 54: Results of string comparison of *optimum* and most similar string

Figure 54 shows the results of a string comparison program on the solution strings of Figure 52 and Figure 53. The program simply compares the bits at each position of each string to see if there is a match. Note that the string from the replacement experiment matches 146 of the 240 bits including sixteen 2^8 bits. In other words, the most similar string of the experiment matches just over sixty percent of the bits and sixteen of the angles are similar (give or take the sign which is reflected in the 2^9 bit). This similarity is further reflected by comparing the twenty-four dihedral angles (translated from the string of 240 bits) of the most similar string of the experiment (see Figure 55) with the dihedral angles of the *optimum* string (see Figure 45).

Residue	ϕ	ψ	ω	χ_1	χ_2	χ_3	χ_4
Tyr:	-177.19	-36.56	-179.65	-178.24	80.16	-74.53	
Gly:	75.59	-97.38	-179.30				
Gly:	-161.37	33.05	-179.30				
Phe:	-162.07	-59.41	-172.97	175.43	70.66		
Met:	-167.34	-73.12	171.91	-172.62	175.43	-100.55	35.86

Figure 55: Dihedral angles of the most similar string of the 100% replacement experiment

Figure 56 provides us with a comparison of the tournament and fitness proportionate selection techniques in the form of a graph showing the three best replacement strategies of each. Notice that the fitness proportionate selection strategies all find lower average minimum energies than the tournament selection strategies. Observe that the best overall method found by the experiments is replacing fifteen percent of the strings while using a fitness proportionate selection operator. While we can not use this graph to *prove* that fitness proportionate strategies are better than tournament selection strategies, we can use the graph as an indicator that fitness proportionate selection might work better with conjugate gradient minimization.

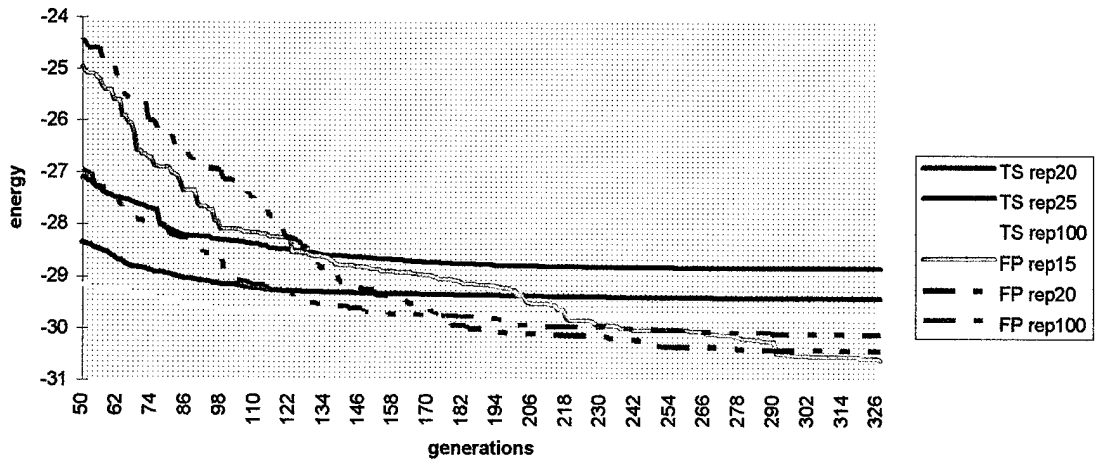


Figure 56: Comparison of tournament and fitness proportionate selection strategies

Another good technique for analyzing the ***nature*** of tournament selection versus fitness proportionate selection is by examining the standard deviation of the energies of all population members at each generation. Viewing Figure 57 (note that it is plotted on a logarithmic scale and calculated on a population of fifty), we see that the standard deviation of the energies in the fitness proportionate generations is more volatile and remains higher. This indicates that really bad solutions (high energy conformations) are being kept in the population. The standard deviations of the energies in the tournament selection generations are much lower which indicates a population of more consistent energies. Now, this consistently low energy population is probably also affected by the Baldwinian replacement strategy. While this discussion does not necessary *prove* anything, it demonstrates the much higher selective pressure of tournament selection when intensified by the Baldwinian strategy.

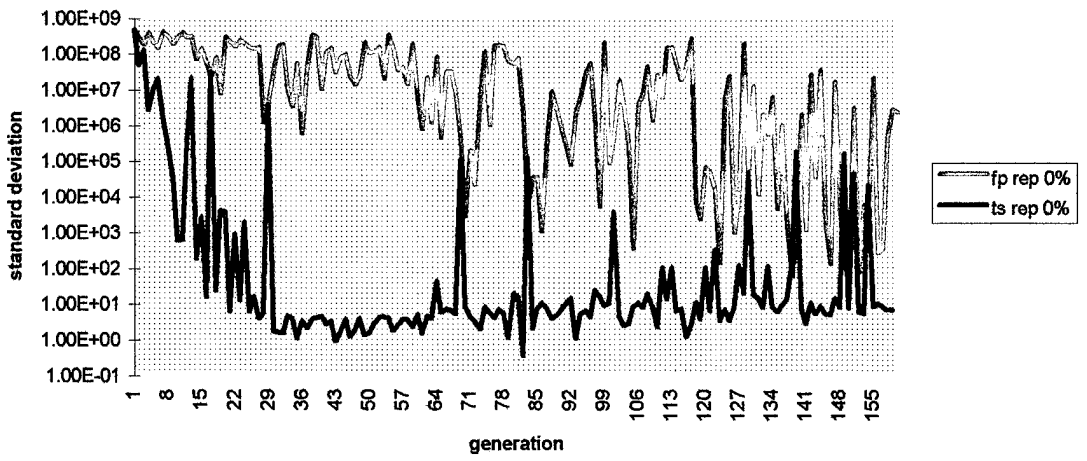


Figure 57: Selection strategy comparison of standard deviations

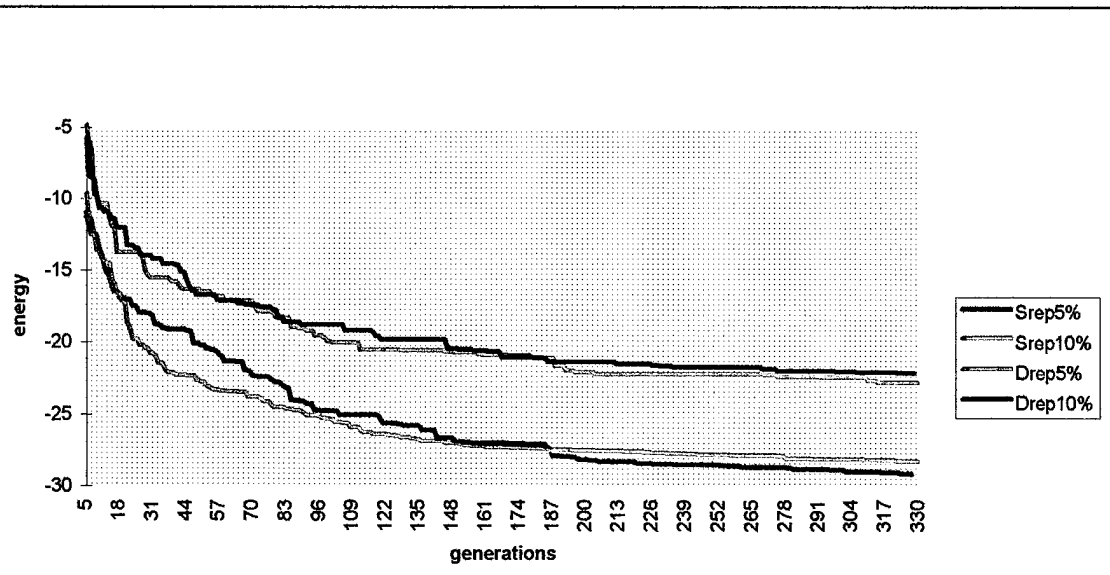


Figure 58: Comparison of String versus Dihedral Replacement

In order to directly compare string versus dihedral replacement strategies, examine Figure 58. This graph represents five run average minimum energies at each generation.

These runs consisted of 12000 trials on a population of fifty individuals. The graph compares the five and ten percent replacement strategies. The dihedral replacement experiment plots begin with *Drep* in the legend while the string replacement plots are indicated with *Srep* in the legend. The graph demonstrates that the replacement of strings is more effective than just replacing a few of the dihedrals (or parts of the strings). Note that the dihedral replacement strategies' average minimum energy is about 7-8 kcal/mol higher than the average minimum energies found with string replacement.

Summary of string replacement experiments

A number of the string replacement application strategies on the average found conformations with energies less than the energy of the *optimum* solution. In fact, a number of applications (see Figure 49 and Figure 51) found energies that were at least ten percent lower than the energy of the *optimum* solution. Moreover, the fitness proportionate application strategy of replacing fifteen percent of the strings found a conformation with an energy of -35.11! In general, the experiments showed the fitness proportionate strategies to be slightly more effective than their tournament selection counterparts. However, the difference is not great enough to discard the idea of using tournament selection (notice in Figure 49 that several application strategies found conformations with lower energy than the *optimum*). In both the fitness proportionate and tournament selection experiments the Baldwinian approach performed poorly enough to indicate that it is not an effective energy minimization tool in a protein folding problem application.

Summary of local minimization experiments

We have observed a number of characteristics of replacement strategies. First, all the string replacement strategies demonstrated potential for both tournament and fitness proportionate selection. The zero replacement strategies do not appear to be as effective in that their average minimum energies are at least 5 kcal/mol higher than the averages of the other string replacement strategies. The dihedral replacement strategies are ineffective when compared with the string replacement strategies. When comparing the most similar string found by the fitness proportionate Lamarckian strategy to the *optimum* string, several of the translated dihedral angles were similar. Several of the string replacement strategies (both tournament and fitness proportionate selection operators) demonstrated their effectiveness as energy minimization tools in this protein folding problem application by consistently finding conformations with similar, and often, lower energies than that of the *optimum* conformation.

Niching Experiments

This set of experiments has the goal of determining the feasibility of applying a sharing strategy (see Chapter IV) to a simple genetic algorithm for protein structure prediction. First, experiments were executed to analyze the behavior of using different numbers of peaks with sharing. Next, experiments were executed in order to observe the effects of string replacement strategies when used in combination with a genetic algorithm that performs sharing at each generation.

Number of peaks experiments

This set of experiments had the purpose of determining the number of peaks to use in the sharing algorithm in finding the best protein structure. In a sense, we can view a simple genetic algorithm as sharing with an infinite number of peaks at which we can cluster solution strings. Because the computation of σ_{share} involves dividing the radius by the 24th (because we have twenty-four dihedrals) root of the number of peaks (see niching discussion in Chapter IV), the test values for the number of peaks were chosen to be 1^{24} (= 1), 2^{24} (= 16777216), and 3^{24} (= 282429536481). This results in relatively different values of σ_{share} (radius/1, radius/2, and radius/3) used in the sharing algorithm. In other words, if we used one, five, and ten peaks for our tests, the twenty-fourth root of each of those are similar and would result in similar values of σ_{share} which would therefore

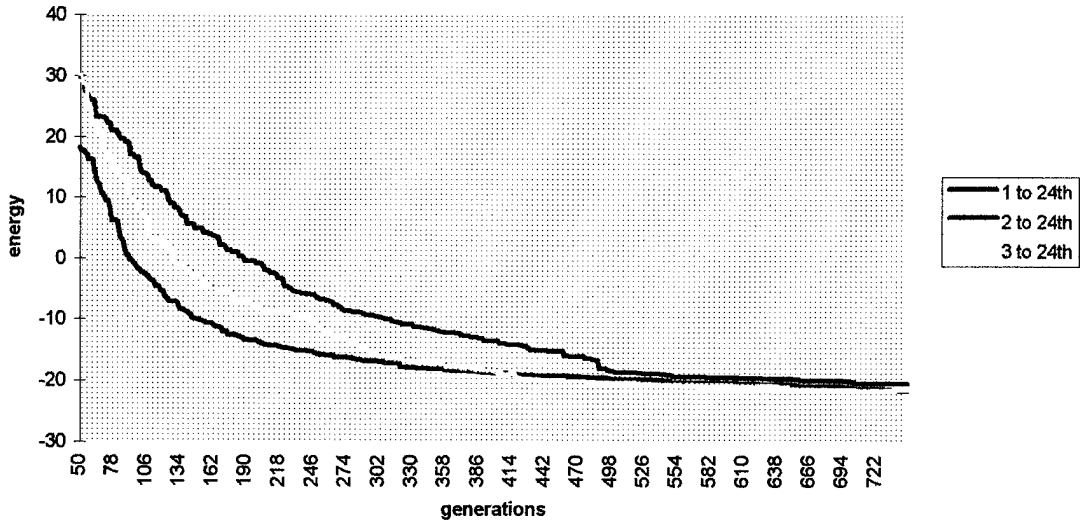


Figure 59: Sharing experiments - number of peaks

generate very similar results. These experiments consisted of ten thousand trials run with a population size of twenty. The smaller population size was chosen in order to facilitate a quicker convergence. These sharing experiments were run on a simple genetic algorithm with a fitness proportionate selection operator. The graph of Figure 59 is of five-run average minimum energies by generation. Notice that all strategies had nearly the same average minimum energy by generation 550. However, careful examination reveals that using the strategies of 1 and 3^{24} peaks worked better than using 2^{24} peaks.

An important concept of niching is that the de-emphasizing of the fitness of clustered solution strings slows down convergence. Figure 60 shows the last three lines of the *out* files of various strategies so we can compare the amount of convergence. Actually, the lines of the *out* files have had some columns removed to show just the columns

GA
```
3993 49988 0.976 -2.245191e+01
3994 49998 0.976 -2.245191e+01
3995 50007 0.978 -2.245191e+01
```

1 to 24th peaks
```
3899 49981 0.961 -1.817938e+01
3900 49995 0.963 -1.817938e+01
3901 50009 0.964 -1.817938e+01
```

2 to 24th peaks
```
3898 49983 0.955 -2.412793e+01
3899 49992 0.952 -2.412793e+01
3900 50005 0.954 -2.412793e+01
```

3 to 24th peaks
```
3765 49989 0.924 -2.473190e+01
3766 49997 0.920 -2.473190e+01
3767 50011 0.928 -2.473190e+01
```

Figure 60: Out files from niching experiments

pertinent to this discussion. From left to right, the first column shows the number of generations, the second column displays the number of trials, the third column is the percentage of convergence, and the fourth column is the minimum energy found at that generation. So, for niching with one peak, after 3901 generations we were 96.4 percent converged with a minimum energy of -18.179 kcal/mol. Notice that the simple genetic algorithm that was run without any niching has achieved over ninety-seven percent convergence which is a greater level than any achieved by the niching strategies. So, we can deduce that the sharing is slowing down convergence as expected.

Notice also that for these particular runs, the niching strategies (2^{24} and 3^{24}_peaks) finds a lower minimum energy than the simple genetic algorithm. In Figure 61, we see the results of experiments that were allowed to run for fifty-thousand trials so we could get a better idea of which strategy may be better. The graph shows five-run average minimum

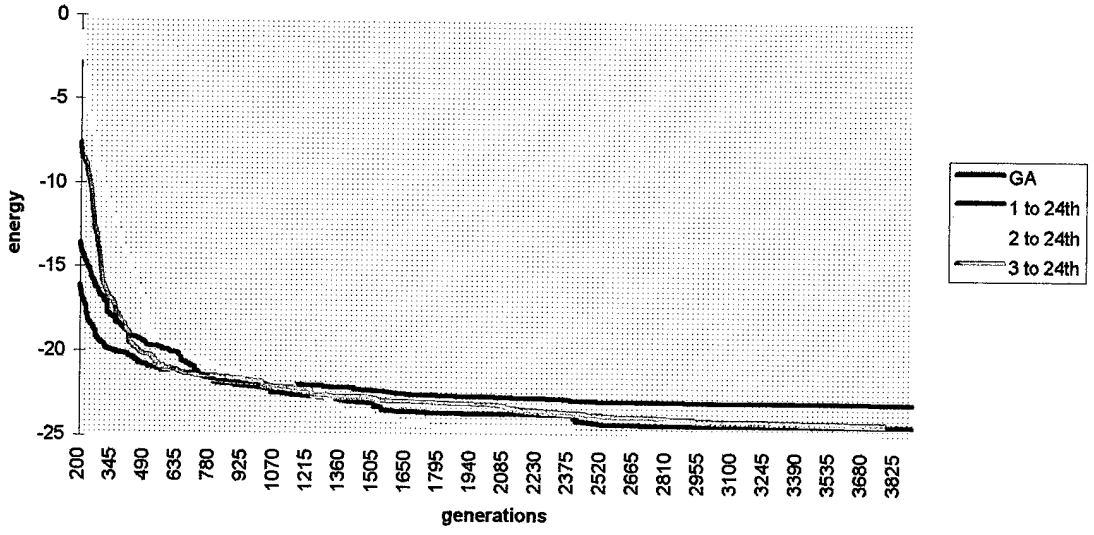


Figure 61: SGA alone versus niching strategies

energies by generation. If we examine the area of eight hundred through eleven hundred (800-1100) generations, we see that the strategies were all averaging about the same minimums with the simple genetic algorithm just barely outperforming the niching genetic algorithm implementations. Observe that as we progress into later generations, the average minimums start to become more distinct. For this set of experiments, the strategy of niching with 2^{24} peaks barely outperforms the simple genetic algorithm (with no niching) and the implementation with 3^{24} peaks.

Figure 62 shows a statistical picture of the final generation of the strategies shown in the graph of Figure 61. It shows the average minimum energy of the final generation, the absolute minimum energy found by each strategy, and the standard deviation of the final generation minimum energies. We can therefore deduce that minimum energies found by the strategy of using niching with 2^{24} peaks were all very similar while there was some variance in the minimum energies found by the strategy of using niching with 1^{24} peaks. Note that the minimum energy found by each strategy is very similar. Moreover, Kruskal-Wallis tests showed that the average minimum energies have no significant differences. So, based on the data of this experiment, it is difficult to conclude (with high confidence) which strategy is better.

	GA	1^{24} peaks	2^{24} peaks	3^{24} peaks
ave	-24.2659	-22.9987	-24.8167	-24.1769
min	-26.7415	-26.7929	-26.0231	-26.6563
dev	1.911804	4.602258	0.99861	2.439931

Figure 62: Niching final generation statistics

String comparisons

Once again, to appraise the success of the experiments, we attempt to determine how close our solutions come to matching the accepted *optimum* conformation. Recall from Figure 45 the dihedral angles (in degrees) of the accepted *optimum* energy conformation of [Met]-enkephalin. The string in Figure 64 (the most similar string of a niching experiment's population) is to be compared with the string of the accepted *optimum* conformation (see Figure 63) to determine, structurally, how similar our solution is to the accepted best conformation.

```
010000101111101111000000001001000100100110111011101111100000
101110111001001011100000011100000111101010001101110000010001
000010111100000101011011100000101010011101000011011010010110
111111001000000000001111001000111101100001010110100000000001
```

Figure 63: Bit representation of *optimum* solution of [Met]-enkephalin

```
010100111011010101011111111111010000000000001101011111111101
111110111110100000100000100000001111111111001010001111111111
001000000000000110111011111001010110010101000111110101001110
111111011000111101110101111000001001000110110010110000000001
```

Figure 64: Most similar string from niching with 2^{24} peaks experiment

Figure 65 displays the results of running a comparison program using the strings of Figure 63 and Figure 64. Note that the most similar solution string only matched 134 bits. If we were to generate a random 240-bit string, we would expect to, on average, have 120 bits in common with the accepted *optimum* string. So, our solution is not much better

than if we had just randomly produced a solution string! Also, observe that we had fifteen 2^9 bits and fourteen 2^8 bits in common. These higher order matchings indicate that we are at least *in the ballpark* of almost two-thirds of the dihedral angles. So, while our complete 240-bit string is rather different than that of the accepted *optimum* conformation, many of our dihedral angles are at least similar. Figure 65 shows the comparison using our best 2^{24} peaks niching solution. When a comparison was performed with the most similar solution string from the 3^{24} peaks experiment and the *optimum* string, there were 137 bits in common with thirteen 2^9 bits and eleven 2^8 bits in common. So, the 3^{24} peaks strategy does find a slightly more similar structure with respect to total bits but with fewer similar dihedral angles.

```
134 bits in common
Order of the bits in common

         15 order (2^9) bits in common
         14 order (2^8) bits in common
         12 order (2^7) bits in common
         14 order (2^6) bits in common
         15 order (2^5) bits in common
         13 order (2^4) bits in common
         12 order (2^3) bits in common
         11 order (2^2) bits in common
         16 order (2^1) bits in common
         12 order (2^0) bits in common

Correlation Matrix
15      0      0      0      0      0      0      0      0      0
12     14      0      0      0      0      0      0      0      0
 8      8     12      0      0      0      0      0      0      0
 7      7      9     14      0      0      0      0      0      0
 6      6      8     10     15      0      0      0      0      0
 3      3      4      5      9     13      0      0      0      0
 2      2      3      4      6      6     12      0      0      0
 1      1      2      2      3      3      6     11      0      0
 1      1      2      2      2      2      4      9     16      0
 1      1      1      1      1      1      2      3      7     12
```

Figure 65: Bit-comparison of the accepted *optimum* and the most similar solution of niching

When comparing the dihedrals of the most similar solution from the niching with 2^{24} peaks experiments (see Figure 66) to the dihedrals of the assumed *optimum* conformation (see Figure 45), it is apparent that several of the *optimal* dihedrals were nearly found in the experiment. This dihedral similarity corroborates our *in the ballpark* conjecture in the bit-comparison discussion of Figure 65.

Residue	ϕ	ψ	ω	χ_1	χ_2	χ_3	χ_4
Tyr:	-62.58	119.88	179.65	-170.51	87.54	-129.02	
Gly:	-90.00	-170.86	178.95				
Gly:	174.02	45.70	-168.75				
Phe:	-90.35	104.06	179.65	-54.49	-79.10		
Met:	-135.00	-47.81	-179.65	-62.58	176.48	-93.16	71.37

Figure 66: Dihedral angles of the most similar niching 2^{24} peaks solution

Niching with string replacement experiments

This set of experiments have the purpose of determining whether or not string replacement is feasible when used with niching. These experiments executed 20000 trials on a population of fifty individuals. We allow the experiments to run with niching for three-hundred generations at which point the population should be well divided into various niches that we wish to explore further. From the three-hundred first generation on, conjugate gradient local minimization is applied using various replacement schemes. In other words, from then on, at every generation niching is performed, followed by the application of genetic operators (see Chapter III), and then the conjugate gradient local minimization steps (see Chapters IV,V) occur. The number of peaks versus string replacement percentages of zero, five, ten, and one-hundred percent are observed.

Figure 67 shows the results of the various replacement percentages when used with niching with 2^{24} peaks. All the experiments have the same average minimum energy until the three-hundredth generation after which the local minimization takes effect. The effect of the local minimization is demonstrated by the forking of the graph. Notice that the one-hundred percent replacement average minimum energy is much lower than that of the other strategies (about -29 kcal/mol versus about -19, -22, and -23 kcal/mol). In other words, the Lamarckian strategy is finding conformations whose average energy is near to the energy of the *optimum* conformation. However, the other experiments indicate that

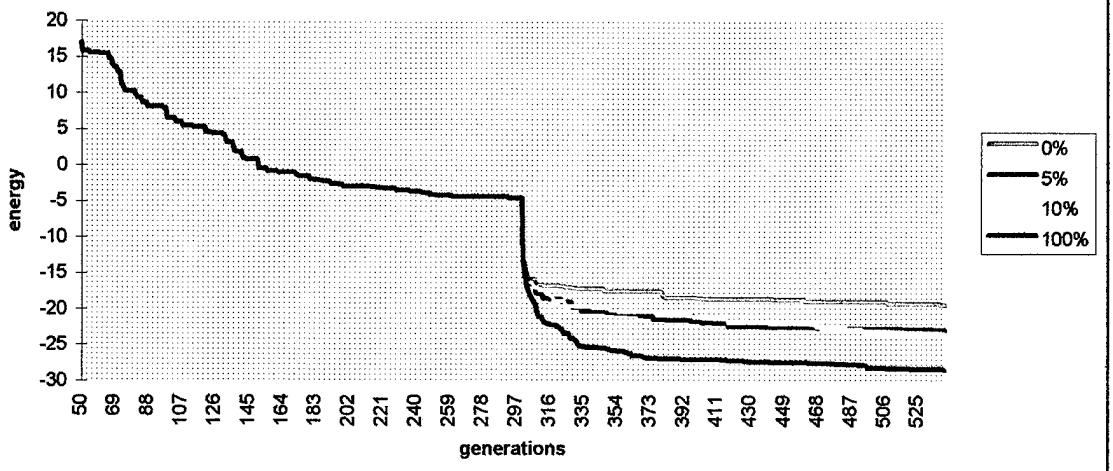


Figure 67: Niching with 2^{24} peaks and delayed string replacement

the combination of sharing with 2^{24} peaks and the other replacement percentages are not effective tools in an energy minimization protein folding problem application. A possible reason behind the apparent ineffectiveness of the other string replacement percentages may be the combination of not reencoding all solutions and then de-emphasizing those

non-encoded solutions' fitnesses. This combination produces the effect of the two strategies more or less canceling each other out!

Figure 68 shows the results of the various replacement percentages used with niching with 3^{24} peaks. All the experiments have the same average minimum energy until the three-hundredth generation after which the local minimization takes effect. This graph also contains the characteristic forking due to the local minimization. Notice that the one-hundred percent replacement average minimum energy is much lower than that of the other strategies (about -31 kcal/mol versus about -23, -25, and -26 kcal/mol). In other words, the Lamarckian strategy is finding conformations whose average energy is lower than the energy of the *optimum* conformation. While all the 3^{24} peaks experiments

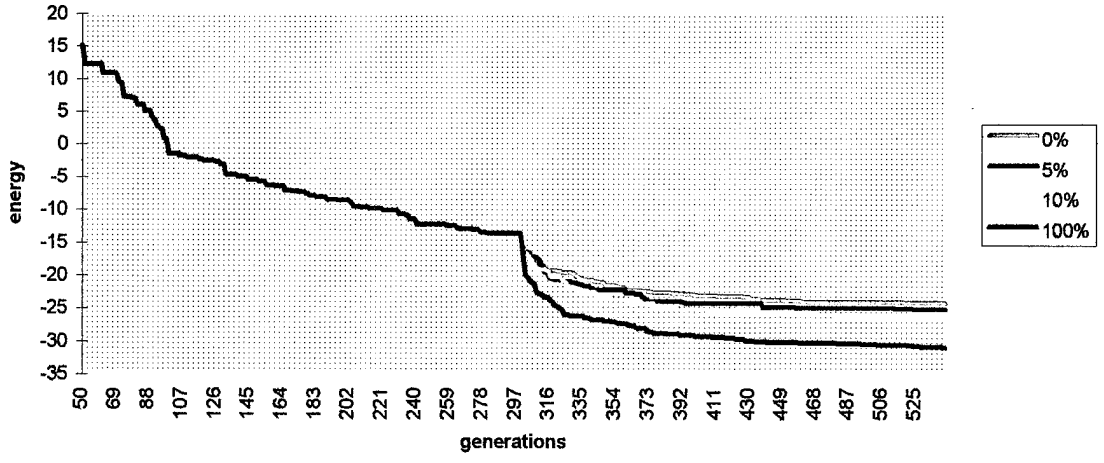


Figure 68: Niching with 3^{24} peaks and delayed string replacement

outperformed their 2^{24} peaks counterparts, the experiments indicate that the combination of sharing with 3^{24} peaks and the replacement percentages (other than the Lamarckian strategy) are not effective tools in an energy minimization protein folding problem application. The 3^{24} peaks strategies outperform the 2^{24} peaks strategies possibly because the 3^{24} peaks strategy results in a greater initial dispersal of the population. This dispersal results in a more complete exploration of the search space. Like the 2^{24} peaks experiments, a possible reason behind the apparent ineffectiveness of the other strategies may be tied to the combination of not reencoding all solutions and then de-emphasizing those non-encoded solutions' fitnesses. So, this combination could likewise be producing the effect of the two strategies canceling each other out.

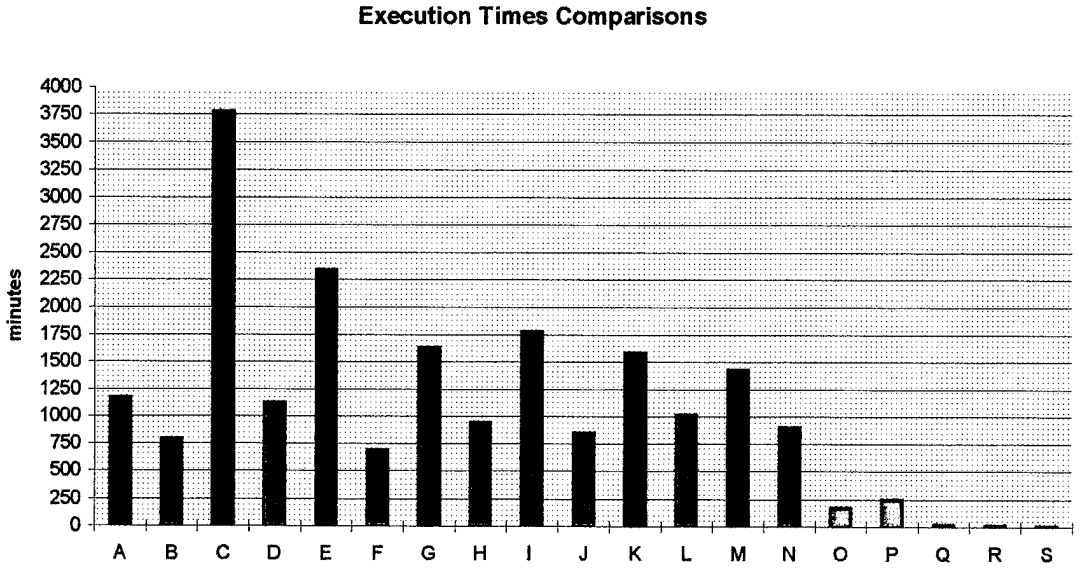
Summary of niching experiments

The niching experiments demonstrated the effects of population dispersal over the fitness landscape. While none of the niching strategies performed especially well by them selves, when niching was applied with the delayed local minimization Lamarckian replacement strategy, some interesting results were produced. For niching with 2^{24} peaks, the Lamarckian replacement produced conformations with average energies of about -29 kcal/mol. In other words, on average we were finding conformations about as "good" as the *optimum* conformation (with respect to energy). For niching with 3^{24} peaks, the Lamarckian replacement produced conformations with average energies of about -31 kcal/mol. This combination of strategies produced the best average energy of any experiment in the thesis effort. This strategy, *on average*, found conformations which had energies five percent lower than that of the *optimum* conformation.

Execution times comparison

In Figure 69, the execution times of various strategies are presented for comparison. It is important to note that these are the ***best*** times achieved for each strategy running on SPARC 20s. Average times would be misleading in this case because students and faculty are constantly logging on and off the machines as well as running their own jobs. A busier machine would drastically slow down execution (especially when jobs are lasting more than a day).

Note that the run times for Q, R, and S are so (relatively) small that they hardly appear on the graph. These experiments only lasted fourteen, twelve, and five minutes respectively. In contrast, the Baldwinian strategy (with fitness proportionate selection) took over 3700 minutes (three days equals 4320 minutes). The conjugate gradient routine loops up to five times in its attempt to approach the local optimum. The Lamarckian strategy more or less *saves* the minimization steps through reencoding the improved strings. Therefore, after several generations, the Lamarckian strategy is using strings which are nearer to local optima and require less of those time-consuming conjugate gradient loops. The Baldwinian strategy, on the other hand, nearly always requires all five loops and so the executions tend to take much longer as demonstrated by the bar graph of Figure 69. Observe that all the fitness proportionate local minimization strategies execute at least a day while their tournament selection counterparts take ten to sixteen hours. So, in some respects, better solution quality (lower energies) seems to have an increasing cost in terms of execution time.



Where:

Fitness Proportionate Selection, population 50, 12000 trials, string replacement

- A - replace 100%
- C - replace 0%
- E - replace 5%
- G - replace 10%
- I - replace 15%
- K - replace 20%
- M - replace 25%

Tournament Selection, population 50, 12000 trials, string replacement

- B - replace 100%
- D - replace 0% (converged after about 5000 trials)
- F - replace 5%
- H - replace 10%
- J- replace 15%
- L - replace 20%
- N - replace 25%

Niching for 300 generations then with local minimization, population 50, 12000 trials

- O - replace 100%
- P - replace 0%

Q is plain niching, population 50, 12000 trials

Simple Genetic Algorithm - no local minimization, population 50, 12000 trials

- R is fitness proportionate selection
- S is tournament selection (converged at around 6000 trials)

Figure 69: Comparison of execution times

Summary

The experiments presented in this chapter are geared toward finding a minimum energy conformation of [Met]-enkephalin. The results of the experiments are used to accomplish the objective of this research effort which is to determine the suitability of using the various local minimization and niching techniques in solving the protein folding problem. The results from the experiments conducted indicate that several string replacement strategies (some with niching) are effective tools in energy minimization protein folding problem applications in that they find conformations whose energies are at or below the energy of the *optimum* conformation. Further analysis and conclusions drawn from these experiments are presented in Chapter VII.

VII. Contributions, Conclusions, and Future Recommendations

Based on the Literature review, design, implementation, and experimental work discussed previously in addition to the observations made throughout this investigation, the major contributions, conclusions and recommendations are presented. The discussion of contributions is broken into the areas of theoretical and application contributions. The conclusions (analytical contributions) are based on the work accomplished in trying to meet the goals of this thesis effort. The goals focus on determining the possible benefits of applying local minimization and niching strategies in conjunction with genetic algorithms for protein structure prediction. These operators are compared by examining the minimum energies and average minimum energies found.

Contributions

The contributions are highlighted in the areas of theoretical contributions and application contributions of this thesis effort. The analytical contributions are discussed in the Conclusions section.

Theoretical contributions

A primary theoretical contribution of this thesis effort is the idea of **locally minimized component replacement** (see Chapter IV as well as Figure 46 and Figure 47). While the concept of replacing percentages of entire solution strings has been discussed in several articles (49, 58), component replacement (though probably already discovered) is not as thoroughly published. In fact, this author has never seen any articles

on the subject so it is possible that it has not been published at all. This is not a safe assumption. (obviously since this author is not well read in *all* journals and publications in the field of evolutionary computation, it would be foolish to lay claim to this concept's discovery).

More important than the issue of who discovered the concept of component replacement, is the fact that this concept deserves further attention despite its relatively poor performance in this application's experiments (see Chapters IV and VI as well as Figure 46 and Figure 47). Component replacement should be tested in applications with different number of components and of varying component length. For example, perhaps it would perform well in an application where the solution is encoded into long strings (many digits) but with just a few components (e.g. five thousand bits and five components) so that encoding a percentage of the components would entail encoding most of the string. On the other hand, perhaps component replacement works by ensuring that at least fifty percent of each string is encoded (in other words, most of the string) regardless of component bit-length.

The next theoretical contribution of this thesis effort is the concept of **combining niching with delayed local minimization** (see Figure 67 and Figure 68) Again, the issue of discovery is not as important as the point that this concept also deserves further attention. The strategy of sharing with 2^{24} peaks combined with delayed (for three hundred generations) Lamarckian replacement found energies comparable to the energy of the *optimal* solution. Also, the strategy of sharing with 3^{24} peaks combined with delayed

(for three hundred generations) Lamarckian replacement found average minimum energies ***five percent less*** than the energy of the *optimal* solution. These results indicate that the prolonged dispersal of a genetic algorithm population over the fitness landscape using niching followed by locally minimizing and then re-encoding every individual for many generations is an effective method of energy minimization. Experiments to measure the effectiveness of this concept in other genetic algorithm applications would be worthwhile.

In summary, this thesis effort has produced two principal theoretical contributions: the concept of component replacement and the concept of niching with delayed local minimization. While niching with delayed local minimization demonstrated the most promise in energy minimization, both ideas deserve more attention and therefore should be tried in different genetic algorithm applications.

Application Contributions

This thesis effort has produced a number of contributions to the AFIT implementation. Most important, is that there now exists a hybrid genetic algorithm software platform. From this, further research can occur on local minimization and niching strategies on [Met]-Enkephalin as well as other protein molecules (with minor software modifications).

In terms of coding, specific contributions to the AFIT implementation are discussed in Appendix D. The appendix details many of the additions and modifications to the AFIT implementation. Other contributions to AFIT implementation are in the form of

documentation. Many of the file and procedure headers were incomplete (or in some cases unchanged duplicates of others). Through this thesis effort, many of those headers have been corrected and completed. Also, commenting within the source code has been added and improved in uncountable areas which allow future programmers to more quickly grasp the idea of what functions the code is performing. Moreover, files have been added in directories to assist researchers. A file called `LM_options_list` which contains the options associated with local minimization and niching (it is similar to the list of Figure 44) has been added to the `~genetic/Toolkit/Simple` directory. In the `~genetic/Toolkit/Messy/Fast` directory, `readme.file_contents` has been added to help would-be fast messy genetic algorithm researchers locate functions and procedures (like the simple genetic algorithm, the fast messy implementation consists of many source files).

Conclusions

The experiments detailed in Chapter VI indicate that some local minimization and niching techniques may be feasible for energy minimization protein structure prediction. Several strategies, on the average, found conformations of lower energy than the accepted *optimum*. However, no experiment found the accepted *optimum* conformation.

The fitness proportionate string replacement strategies performed better than their tournament selection counterparts (the twenty percent replacement experiment achieved a minimum energy of -35 kcal/mol). There are replacement strategies using both selection operators that found conformations with average minimum energies below the energy of

the *optimum* conformation. The Baldwinian replacement strategy performed poorly when used with either selection operator and so is an ineffective tool in this energy minimization protein folding problem application. See Figure 70 for a brief summary of the experimental results.

Strategies of arbitrarily replacing a percentage of the dihedrals (or parts of the strings) performed poorly when compared with the results of total string replacement for both fitness proportionate and tournament selection. Based on these poor results, this strategy (see the Theoretical Contributions discussion on pages 87-88 for further insight) does not seem feasible for a protein folding application. However, this type of strategy could be useful for other applications and so should not be discarded totally. See Figure 70 for a brief summary of the experimental results.

Niching strategies did not perform as well by themselves but show great promise when combined with delayed local minimization Lamarckian strategies. The concept of allowing the solution space to be firmly divided into niches *and then* applying local minimization (and encoding all strings) outperformed every other strategy tested in this thesis effort (in terms of average minimum energy). Further experimentation should be applied to the concept of diversifying the solution space using niching combined with the exploitation of the solution space using conjugate gradient local minimization (see the Theoretical Contributions discussion on page 88 for further insight). See Figure 70 for a brief summary of the experimental results.

Strategy	Average	Minimum
Fitness proportionate Lamarckian replacement	-30.1201	-32.8813
Fitness proportionate 20% string replacement	-30.4155	**-35.1889**
Fitness proportionate 15% dihedral replacement	-21.0916	-22.5633
Tournament 5% dihedral replacement	-24.4519	-27.1842
Tournament Lamarckian replacement	-29.7675	-31.3162
Tournament 20% string replacement	-29.4104	-32.093
Niching(3^{24}) with delayed Lamarckian replacement	**-31.0592**	-32.7731
Niching with 3^{24} peaks	-24.1769	-26.6563

Figure 70: Comparison of energies found by the various strategies (best are highlighted)

In terms of execution times (see Figure 69), most of the strategies were finished in 48 hours. Compared to the possible *two years* of laboratory time of the physical techniques (see Chapter II), these quality solutions were obtained in only about *two days*. So most of the strategies when considering execution time and solution quality, are effective in this protein folding problem application. However, the Baldwinian runs frequently took more than seventy-two hours which coupled with their poor solution quality indicates that they are impractical for this protein folding problem application.

Future Recommendations

There are a number of possible techniques to try in our search for better ways to find optimal conformations. If anything, this thesis effort demonstrated the *potential* of applying a local minimization technique with genetic algorithms in polypeptide energy minimization. So, the local minimization techniques should now be incorporated into the existing AFIT serial fast messy genetic algorithm code and thoroughly tested. Next, the

local minimization code should be inserted into the *parallel* versions of the simple and fast messy genetic algorithm. The results of experiments conducted with these implementations can be compared with the results presented in this thesis to determine which methods are most worthy of further testing (see Appendices A and B for additional information on parallel/distributed computing and messy genetic algorithms).

Because some of the niching implementations showed such promise, the niching code should be combined with the parallel simple genetic algorithm code. The combination of niching (which spreads the population out over the solution space) and parallel computing (which spreads the population over the different nodes) could yield interesting results (see Appendix A for information on parallel/distributed computing).

There are several other methods that could also be applied. Research should be applied toward using real-valued encodings and operators applied to the protein folding problem while paying attention to performance (in terms of both solution quality and execution times). Also, trying other types of selection operators could offer benefits. Another interesting experiment would be to adapt the current fitness proportionate operator so that it applies a higher selective pressure by periodically eliminating members with poor fitness (poor fitness could be defined as being more than three deviations from the mean fitness, for example).

Finally, there are a few software engineering concerns. First, the AFIT/WL implementation is currently modified by a number of researchers. There needs to be established a system for communication about code changes between the researchers. One solution may be for each researcher to establish one's own working directory in which coding and testing is accomplished. Then, after group approval, the code could copied into the main implementation. Also, there need to be some standards defined for commenting (both inside the code and in headers). The existing code is very well documented in some areas while not at all in others. The issue of *what is a useful, complete comment* needs to be addressed and agreed upon.

Summary

This chapter summarizes the general conclusions that can be derived from this investigation. These conclusions are used to indicate possible areas of future research. Overall, this thesis documents the results of various applications of local minimization strategies and niching strategies to the AFIT genetic algorithm implementation.

Appendix A - Parallel/Distributed Computing

This appendix summarizes current knowledge of parallel/distributed computing techniques with emphasis placed on the possible benefits of combining them with a genetic implementation towards solving the protein folding problem. First, this chapter discusses parallel computing paying attention to issues such as scalability and the isoefficiency function. Then, this chapter addresses distributed computing focusing on issues, PVM, and MPI.

Parallel Computing

When you have to dig a ditch, if you have a helper start at one end while you start at the other, then the task is accomplished much quicker than by you working alone. This is the same philosophy that is used is parallel computing. Frequently, a job can be accomplished much quicker by dividing tasks among multiple processors. An important consideration in parallel computing is communication - all the processors need to know what is going on, what to do with their results, and then need to send those results. In our ditch-digging example, the best communication scheme would probably be to initially give our helper all the necessary information: where to start, how deep, how wide, and so on. If the helper keeps having to run the full length of the ditch to ask you questions, it lowers the helper's productivity, your productivity, and as a result, the overall productivity. However, if our helper has a limited memory capacity and only can remember a few things (how bright can a ditch-digger's assistant be?) then we might have to adopt a different

communications scheme. Similarly, in parallel computing, we also have to take local memory and message sending time into account.

Frequently, a job can be accomplished much quicker by dividing tasks among multiple processors. An important consideration in parallel computing is communication — all the processors need to know what is going on, what to do with their results, and then need to transmit those results. In parallel computing, we also have to take local memory and message sending time into account. (34)

Massively parallel computers (computers having a large number of processors) can have over a thousand processors, and plans are being drawn for architectures with more than one million nodes. Parallel solutions are said to be ***scalable*** if additional processors can be used efficiently. (34) This is important because after some point our job can actually be **slowed down** if we add additional processors. In our ditch-digging example, we can only use a limited number of additional assistants before they start getting in the way of each other and slowing down the job. So, at first, our ditch job is *scalable*. However, after reaching one assistant per few feet of ditch, additional assistants are not effective and, in fact, could be detrimental. In comparison, to reap the benefits of parallelism, we are looking for algorithms that are scalable.

How effective is parallel processing? The potential gains of parallelism are made very apparent with the recent announcement that Sandia National Laboratory achieved 281 billion floating point operations per second (gigaFLOPS) on two hyperlinked Intel

Paragons (6768 processors in parallel) using the Linpack Benchmark and 328 gigaFLOPS using electromagnetic radar signature calculation code. This is made more dramatic when you consider that each of the 3384 nodes were actually just a pair of Intel i860 XP processors which are each capable of a mere 50 million floating point operations per second (megaFLOPS). (52)

As stated previously, we can view a system's scalability by using its isoefficiency function. For example, say we have p processors, a problem size of W, and the total time on all processors that it takes to solve a given problem is pT_p.. Out of pT_p ,we spend only W units of time performing useful work. We can now express the overhead (T_o) function. Then, we can derive the isoefficiency function as follows:

$T_o = pT_p - W$ (overhead function)

$T_p = [W + T_o(W,p)] / p$ (solving for T_p)

$S = W / T_p$ (speedup)

$= W*p / [W + T_o(W,p)]$

$E = S/p$ (efficiency)

$= W / [W + T_o(W,p)]$

$= 1 / [1 + T_o(W,p)]/W$

$W = E/(1-E) * T_o(W,p)$ (solving for W)

let constant $K = E/(1-E)$ depend on the maintained efficiency

So, $W = K * T_o(W,p)$ (isoefficiency function)

Figure 71: Derivation of the Isoefficiency Function (34)

The isoefficiency function is telling us the difficulty (or lack thereof) with which a parallel system can keep a constant efficiency and so achieve some speedup in proportion to the number of processors. We hope for a small isoefficiency function because that indicates that we only need small increments in the problem size for the efficient use of more processors. In other words, we would have a highly scalable system. (34)

The main reason that we are interested in parallel computing is that genetic algorithms are easily parallelized and very scalable. One approach puts multiple copies of the same program on each processor, starts their execution with different seeds for the random number generators, and selects the best solution after all processors have finished. Another approach (referred to as the *island model*) is where the population is divided up into subpopulations which are grouped on individual processors which run independent genetic algorithms. This results in little communications overhead but at a possible sacrifice in solution quality. (17, 19)

Distributed Computing

As personal computers (PCs) become more powerful and less expensive (more CPU per dollar), we are looking for ways to divide jobs among groups of PCs to reap parallel benefits. This type of computer task division is known as distributed computing. Distributed computing is not limited to just networks of PCs. It can be used in a network of any type of systems (e.g. SPARC 20 workstations). Some of the characteristics of a distributed system include the lack of a shared clock and the lack of shared memory.

There are a number of strategies for controlling the network. The two methods discussed are *Parallel Virtual Machine* (PVM) and *Message Passing Interface* (MPI). (34, 48, 54)

PVM

PVM allows a heterogeneous collection of UNIX systems to be viewed by a user's program as a single parallel *virtual* computer. PVM was developed at the Oak Ridge National Laboratory by Vaidy Sunderham and Al Geist. The initial version was a prototype used only in the lab. After a period of testing, version 2 was written and released through the University of Tennessee. As of 1994, version 3.3 had been developed and released. PVM works by viewing the user's application as a set of cooperating tasks. PVM manages the initialization, termination, and synchronization of these tasks. Communication is handled through *primitives* which involve strongly type constructs for buffering and transmission. Those constructs includes those for sending, receiving, broadcasting, barrier synchronization, and global summing. PVM allows tasks the ability to start and stop other tasks, and to add or delete computers from the virtual system. PVM is not limited to distributed computing as it can be used with massively parallel machines as well. (22)

MPI

The Message Passing Interface standard specification was completed in 1994. Its goal was to develop standard syntax and semantics of massage passing routines (in FORTRAN or C) which would allow for portability. MPI is easily compatible with

distributed-memory multicomputers and shared-memory multiprocessors. The MPI standard was developed over a year of intensive meetings involving over eighty people from approximately forty organizations, many vendors of concurrent computers, and researchers from universities, government laboratories, and industry. Their combined efforts resulted in the publication of the MPI specification. MPI is still in relatively early development. The next version of MPI is expected to include provisions for the following: Parallel I/O, Remote store/access, Active messages, Process startup, Dynamic process control, Non-blocking collective operations, FORTRAN 90 and C++ language bindings, and Graphics. Real-time support MPI can be used as a communications layer built on the hardware platform which allows PVM to be ported to MPI to exploit vendor-supplied communication performance. (14, 22)

Issues of Distributed Computing

A number of factors come into play when dealing with distributed computation. First, there is *granularity*. Granularity is the ratio of uninterrupted computation time to communication operations. This should not be confused with parallel granularity which is the ratio of the power of the processors versus the number of processors. Another issue is *coupling*. Coupling is the amount a process depends on companion processes for the overall computation to succeed. Another issue is portability. Portability is the aspect of a system component that allows it to be used in various environments. For instance, software portability would indicate the extent that software can be ported from one hardware system to another. Another issue is cache coherence or more generally, *data*

coherence. Data coherence is the problem that arises when one processor changes the values of data in its local memory. This results in the data located in shared memory and the data in the local memory of other processors becoming obsolete. One technique for handling the problem is to have the processor write to a shared location which is then used to update all memory locations. (34, 41, 48)

Summary

Parallel and distributed computing can offer us a large advantage in problem solving. It enables us to divide our problem into concurrent tasks and solve the problem faster. Key issues in parallel/distributed computing such as efficiency and overhead contribute to the concept of scalability. Scalability is what provides us that large advantage in problem solving. Fortunately, genetic algorithms are *scalable*.

Appendix B - Messy Genetic Algorithms

Messy genetic algorithms were developed largely to overcome the problem of deception. The messy genetic algorithm combats deception through the use of *partially enumerative initialization* (PEI). In PEI, the initial population of possible building blocks (partial solutions) is created with each being a specified length. With a block size of n, the initial population size is equal to:

$$2^n \binom{l}{n} = 2^n \frac{l!}{(l-n)!n!}$$

Figure 72: MGA population size with string-length (l)

This can result in populations much larger than those used by simple genetic algorithms. Note that if we set the block size equal to the string length ($n = l$) then our population size is equal to just 2^l which is the same as the initial population size of a simple genetic algorithm (using binary digits). This is logical since we would be manipulating fixed-length blocks that encompass the entire string just as a simple GA manipulates the entire fixed-length string. Moreover, if we set the block size equal to one, then our initial population size would be $2l$. In other words, if our block is made up of just one bit, then it would only take $2l$ strings to cover the possible values. (17, 25, 43, 44)

Another key difference between messy and simple genetic algorithms is that messy GAs encode both the string position (locus) and its value (allele) in variable-length strings. These strings are built up to allow a genetic algorithm to cover all features of a problem.

Messy Genetic algorithms tend to mimic nature in that over time simple structures develop into complex ones. In doing so, they allow the existence of *under-specified* and *over-*

```
Perform PEI
evaluate fitness

for i=1 to max. number of primordial generations
        perform tournament selection
        periodically half the population (e.g. every 10 iterations)

for i=1 to max. number of juxtapositional generations
        perform cut-and-splice
        evaluate fitness
        perform tournament selection
```

Figure 73: Messy Genetic Algorithm

specified strings (hence the variable lengths). Under-specified strings do not have an allele for every locus. A locally optimal competitive template is used to supply the values for the unspecified loci. Over-specified strings have more than one allele per locus. In this case, the locus is set to the value encountered **first**. (17, 25, 26)

The messy genetic algorithm consists of two phases— the Primordial Phase and the Juxtapositional Phase. The word *primordial* implies happening or existing first. The primordial phase happens before the juxtapositional phase and so hence the name. In the Primordial phase, we are concerned principally with enriching the population with above average building blocks. This is usually accomplished through tournament selection. The other main purpose served by the primordial phase is the reduction of the population size to a level (usually halved) that enables the juxtapositional phase to operate efficiently and effectively on the population. (2, 17, 25, 43, 44)

The Juxtapositional phase is similar to a simple genetic algorithm. The word *juxtapositional* means positioned side by side which makes for a logical name since we are placing strings side by side for comparison. The main difference between the juxtapositional phase and the simple genetic algorithm is that we are now dealing with variable length strings. So, the crossover operator is replaced with the Cut-and-Splice operator. The Cut-and-Splice operator picks random points on parent strings and cuts off the ends and splices the end onto the other string head to form the children of the next generation (very similar to crossover except we are *not* cutting off equal-length ends from the parent strings). The mutation operator is generally not used with messy genetic algorithms. (4, 17, 24, 25, 43, 44)

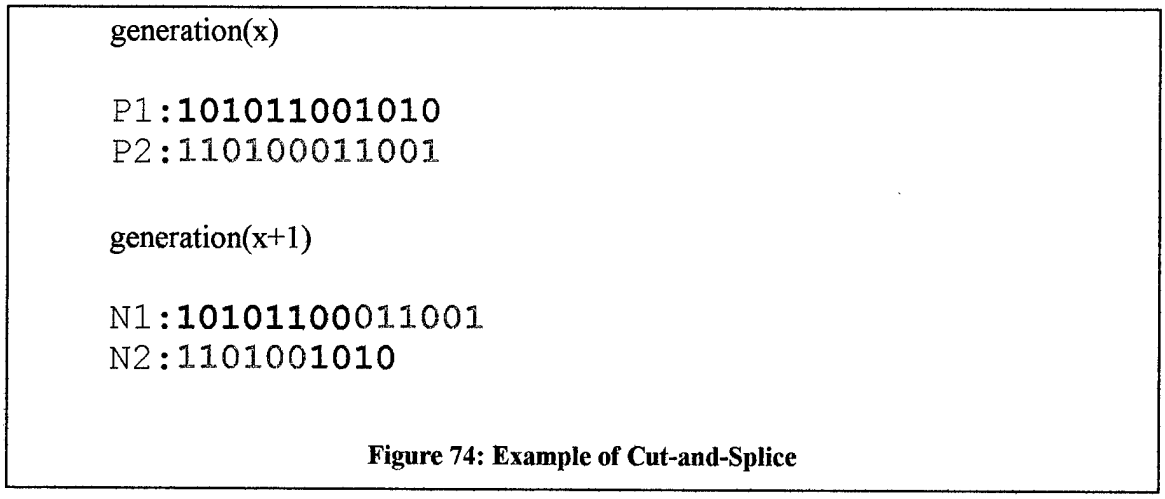


Figure 74: Example of Cut-and-Splice

There are several advantages of the messy genetic algorithm. First, it handles the problem of deception primarily by finding tightly-coded building blocks and then finding globally optimal structures by juxtaposing those building blocks. This, in part, affects the next advantage which is generally better solution quality. However, the messy genetic

algorithm has a big disadvantage in that the increased population size causes more computations which leads, in some cases, to dramatic increases in execution times. To overcome such problems, the fast messy genetic algorithm was developed. (4, 17, 24, 25, 26, 43)

The Fast Messy Genetic Algorithm

What makes a fast messy genetic algorithm *fast*? The fast messy genetic algorithm is very similar to the messy genetic algorithm but with a few key differences. The first and principal difference is in the initialization. The fast messy genetic algorithm reduces the complexity of the initialization phase (messy GA $\rightarrow O(l^k)$ versus the fast messy GA $\rightarrow$ $O(l \log l)$) which, in turn, reduces the overall algorithm time and space complexity. The fast messy GA uses *probabilistically complete initialization* (PCI) which creates a population whose size is equivalent to the population size at the end of the primordial phase of messy genetic algorithms. The fast messy GA then enriches the population through alternating steps of tournament selection and *building block filtering* (BBF). The tournament selection increases the percentage of individuals containing building blocks and then BBF randomly deletes some number of genes from every individual. That number is chosen so that many of the building blocks are disrupted (but not all!). The end result is a population of partial strings that have a high expected proportion of building blocks. The last key difference is that there is more conservative thresholding in the tournament selection of the fast messy GA compared to that in the messy GA. (17, 44)

```
Perform PCI
evaluate fitness

for i=1 to max. number of primordial generations
        perform tournament selection
        if a BBF is scheduled then
                perform BBF
                evaluate fitness

for i=1 to max. number of juxtapositional generations
        perform cut-and-splice
        evaluate fitness
        perform tournament selection
```

Figure 75: Fast Messy Genetic Algorithm

Even though they generally have lower execution times than the messy genetic algorithm and offer increased solution quality versus the simple genetic algorithm, fast messy genetic algorithms have not proven to be the end-all solution to our problems. The principal problem is that the best parameters for the fast messy genetic algorithm are presently unknown. In some cases, the fast messy genetic algorithm has been shown to have much greater execution times than the simple genetic algorithm. Until better fast messy parameters are found, previous AFIT research indicates that the simple genetic algorithm (on a parallel platform) is the preferred technique for the protein folding problem. (19, 20, 21)

Fast Messy Genetic Algorithms and Local Minimization

Figure 18 shows the possible locations of a local minimization step in a fast messy genetic algorithm. Having a local minimization step in all three locations could stand to provide us the most benefit from local minimization but would suffer with respect to

computation time. Performing the local minimization step only within the Primordial loop would allow us to generate a highly fit population for the juxtapositional loop to operate on. On the other hand, there is logic to a strategy of sending generic building blocks (ones

```
Perform PCI

evaluate fitness

Local Minimization step

for i=1 to max. number of primordial generations
        perform tournament selection
        if a BBF is scheduled then
                perform BBF
                evaluate fitness
                Local Minimization step

for i=1 to max. number of juxtapositional generations
        perform cut-and-splice
        evaluate fitness
        Local Minimization step
        perform tournament selection
```

Figure 76: Fast Messy Genetic Algorithm with Local Minimization

that have not been influenced by local minimization) to the juxtapositional loop which contains a local minimization step. This configuration would make the genetic algorithm rapidly approach a minimum. The question is, *would our solution quality suffer?* This and other questions (about solution quality versus execution time) leaves no doubt that future research is needed in determining the best strategy for placement of local minimization steps in a fast messy genetic algorithm.

Appendix C - Altering the replacement percentages

This appendix describes the steps required to modify the different string replacement percentages when using local minimization in the AFIT implementation. Most of the options of the implementation are set using values and flags in the user-defined *in* file (see Figure 43). By setting just the local minimization flag (m) in the options line, the implementation defaults to the Baldwinian strategy (zero percent replacement). If the *Lamarckianflag* (E option) is included in the options, then one-hundred percent of the strings are replaced in each generation's local minimization step. However, to set the percentage of replacement of strings (or components) to a value other than zero or one-hundred, then the *Davisflag* (Z option) must be included in the options line and the actual code must be modified to indicate the desired percentage.

The local minimization code that must be modified is located in the last section of the file, `energy.c`, which is located in the `~genetic/Toolkit/CHARMm` directory. The code (see Figure 77) contains a section for string replacement (starts immediately after "`Lamarckian or Davis's replacement evolution`" comment) and a section for dihedral replacement (which is commented out). To change the percentage of replacement, change the `rand` number in the `if` statement (note that it is currently set to .10 or ten percent replacement in line 8). Finally, observe that the if block also handles the *Lamarckian* (one-hundred percent replacement) flag.

```
if(Minimizationflag)
      {

      frprmn(P, num_dihedrals, 0.1, &dummy, &energy, func, dfunc);

      /**********Lamarckian or Davis's replacement evolution*********/

     if ((Lamarckflag) || ((Davisflag) && (Rand() < 0.10)))
         {
         start = 0;

         for (i = 0; i < num_dihedrals; i++)
           {
            tempint = (unsigned long)(((P[i+1] + PI)/twoPI) *
max_range);
            Itoc (tempint, &buff[start], slice);
            start = start + slice;
           }

         }    /*if Lamarckian or Davis*/


      /************Dihedral replacement code**************/

      /* if (Davisflag)
         {
         start = 0;

         for (i = 0; i < num_dihedrals; i++)
           {

            if (Rand() < 0.10)
              {
               tempint = (unsigned long)(((P[i] + PI)/twoPI) *
max_range);
               Itoc (tempint, &buff[start], slice);
               start = start + slice;
              }
           }
         }  */   /*if Davisflag*/


      return(energy);

      }        /*if Minimization*/
```

Figure 77: Minimization segment of energy.c source file

Appendix D - Listing of Implementation Modifications/Additions

This appendix itemizes many of the modifications and additions to the AFIT implementation that were involved with this thesis. It is important to note that most all of the actions described in this appendix were accomplished in a "team" environment and so several individuals (42) were involved. This appendix comments on the actions which were especially labor-intensive to the author.

Code modifications in ~genetic/Toolkit/CHARMm for Thompson's (56) transformation

File	Action
molecule.h	modified ATOM_TYPE structure by replacing declaration of coords[3] with declaration of transmat[4][4]; added Btransmat[4][4] structure;
molecule.c	replaced references to coords with equivalent transmat notation; coded identity matrix for atom#1; coded B-matrices for atoms 2 and 3 after hand-calculating those values; coded known values used for first (bond angle terms) and fourth rows (1 0 0 0) of the all the B-matrices;
new_coordinates.c	removed old coordinate computations; added procedure Mat_x_Mat which computes A-matrices using second and third rows of B-Matrices; added code to handle *atom 42 problem*;

Coding in ~genetic/Toolkit/CHARMm for conjugate gradient minimization

File	Action
derivative.c	implemented Thompson's (56) derivative algorithm to calculate the partial derivative representing the change in position with respect to the change in the dihedral;

	implemented code to calculate the partial derivative representing the change in distance with respect to the change in the position; implemented code to calculate the partial derivative representing the change in energy with respect to the change in the interatomic distance; implemented code to multiply partials resulting in the derivative of the non-bonded energy with respect to a particular dihedral; partial derivative code segments were placed in a loop structure to generate an array containing derivative of the non-bonded energy with respect to all dihedrals; the author did many of the hand calculations and much of the initial coding which was followed up by editing and additions by other individuals (42);
`frprmn.c`	modified from Numerical recipes in C (51) for use with the AFIT implementation;
`dbrent.c`	modified from Numerical recipes in C (51) for use with the AFIT implementation;
`nrutil.c`	modified from Numerical recipes in C (51) for use with the AFIT implementation;
`nrutil.h`	modified from Numerical recipes in C (51) for use with the AFIT implementation;
`mymnbrak.c`	Gates(42) created this modification of the `mnbrak.c` code from Numerical recipes in C (51);
`dlinmin.c`	modified from Numerical recipes in C (51) for use with the AFIT implementation;
`energy.c`	added calls containing calculated derivatives and energy function to `frprmn.c` which return minimized function.

Coding activities in `~genetic/Toolkit` for local minimization strategies

File (Subdirectory)	**Action**
`energy.c (CHARMm)`	added code to partially or completely encode (strings and components of) locally minimized solutions (see Figure 77); added various flag declarations;

`generate.c (Simple)`	added constructs to activate and deactivate flags which correspond to strategies that depend on number of generations completed; added printf statements to force output in the most usable format for this thesis effort;
`input.c (Simple)`	added in options and corresponding flag assignments for the various local minimization application strategies;
`format.h (Simple)`	added flag declarations for the various local minimization application strategies;
`global.h (Simple)`	added flag declarations for the various local minimization application strategies;

Coding activities in `~genetic/Toolkit/Simple` for niching strategies

File	**Action**
`input.c`	added niche flag option and flag assignment; modified *in* file format to include number of peaks assignment;
`format.h`	added flag declaration for the niching strategy; added declaration for number of peaks variable;
`global.h`	added flag declaration for the niching strategy; added declaration for number of peaks variable;
`select.c`	added call to niche procedure; modified roulette computations to accommodate the de-emphasized fitnesses resulting from niching;
`niche.c`	implemented sharing algorithm (12,28); the author did much of the initial coding which was followed by editing and additions by other individuals (42);

Bibliography

[1]. Adler, Dan, Genetic Algorithms and Simulated Annealing: A Marriage Proposal, NY, NY, 1993.

[2]. The American Heritage Dictionary, 2nd College ed., Boston, MA: Houghton Mifflin Company, NY, NY: Dell Publishing Company, 1986.

[3]. Brassard, Bratley, Algorithmics, Theory and Practice, Englewood Cliffs, NJ: Prentice Hall, 1988.

[4]. Brinkman, Chase, Gates, Gordon, Olsan, Merkle, Lamont, Compendium of Parallel Programs for the iPSC Computers, Vol. V, version 2.0, *Evolutionary Algorithms* - Supplemental text for CSCE656, Department of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, May 26, 1994.

[5]. Brinkman, Genetic Algorithms and their Application to the Protein Folding Problem, MS thesis, AFIT/GCE/ENG/93D-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1994.

[6]. Brooks, Bruccoleri, Olafson, States, Swaminathan, Karplus, Journal of Computational Chemistry, Vol. 4, No. 2, 1983, John Wiley and Sons Inc., *CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations.*

[7]. Chan, Hue Sun and Dill, Ken A., The Protein Folding Problem, *Physics Today*, pp. 24-32, February 1993.

[8]. CHARMm, 1992. Parameter file for CHARMm version 22.0, Molecular Simulations Incorporated.

[9]. Christofides, Graph Theory, An Algorithmic Approach, Academic Press, 1975.

[10]. Cormen, Leiserson, & Rivest, Introduction to Algorithms, Cambridge MA: MIT Press, 1990.

[11]. Conte, Samuel D. and de Boor, Carl, Elementary Numerical Analysis: An Algorithmic Approach, 3rd ed., NY, NY: McGraw-Hill Publishing, 1980.

[12]. Deb, Kalyanmoy and Goldberg, David E., *An Investigation of Niche and Species Formation in Genetic Function Optimization*, Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 42-50, 1989.

[13]. DeJong, Kenneth A, An Analysis of the Behavior of a Class of Genetic Adaptive Systems, PhD dissertation, The University of Michigan, Ann Arbor, MI, 1975.

[14]. Dongarra, J.J. & Otto, S.W. & Snir, M. & Walker, D.W., *An Introduction to the MPI Standard*, Communications of the ACM (submitted), January 1995.

[15]. Finney, Ross L. and Thomas, George B., Calculus, Reading, MA: Addison-Wesley Publishing Company, 1990.

[16]. Fogel, Gary B., *An Introduction to the Protein Folding Problem and the Potential Application of Evolutionary Programming*, The Second Annual Conference on Evolutionary Programming, pp. 170-177, San Diego, CA: Evolutionary Programming Society, 1993.

[17]. Gates, George H. Jr., Predicting Protein Structure Using Parallel Genetic Algorithms, MS thesis, AFIT/GCS/ENG/94-D, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1994.

[18]. Gates, George H. Jr., `mymnbrak.c`, code contained in Genetic Algorithm Toolkit ver 2.0 (36), June 1995.

[19]. Gates, Pachter, Merkle, Lamont, Simple Genetic Algorithm Parameter Selection for Protein Structure Prediction, WL/MLPJ, Wright-Patterson AFB, OH.

[20]. Gates, Pachter, Merkle, Lamont, Parallel Simple and Fast Messy GAs for Protein Structure Prediction, WL/MLPJ, Wright-Patterson AFB, OH, January 17, 1995.

[21]. Gates, Pachter, Merkle, Lamont, Parallel Simple GAs Vs Parallel Fast Messy GAs for Protein Structure Prediction, WL/MLPJ, Wright-Patterson AFB, OH, May 1995.

[22]. Geist et al, PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing, Cambridge, MA: MIT Press, 1994.

[23]. Goldberg, David E. Genetic Algorithms in Search, Optimization, and Machine Learning, Reading, MA: Addison-Wesley Publishing Company, 1989.

[24]. Goldberg, David E., et al, *Genetic Algorithms and Walsh Polynomials: Part I, A Gentle Introduction*, Complex Systems, 3:129-152 (1989).

[25]. Goldberg, David E., et al, *Messy Genetic Algorithms: Motivation, Analysis, and First Results*, Complex Systems, 3:493-530 (1989).

[26]. Goldberg, David E., et al, *Messy Genetic Algorithms Revisited*, Complex Systems, 4:415-444 (1990).

[27]. Goldberg, David E., *Making Genetic Algorithms Fly - A Lesson from the Wright Brothers*, seminar given at the Air Force Institute of Technology, August 23, 1995.

[28]. Goldberg, David E. & Richardson, Jon, *Genetic Algorithms with Sharing for Multimodal Function Optimization*, Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms, pp. 41-49, 1987.

[29]. Grant, *An introduction to Genetic Algorithms*, C/C++ Users Journal - Advanced Solutions for C/C++ Programmers, Vol.. 13, Num. 3, March 1995.

[30]. Grefenstette, John, A User's Guide to Genesis 5.0, Technical Report, Nashville, TN, Vanderbilt University, 1990.

[31]. Holland, Adaptation in Natural and Artificial Systems, The University of Michigan, Ann Arbor, MI, 1975.

[32]. Judson R.S. and McGarrah M., *Analysis of the Genetic Algorithm Method of Molecular Conformation Determination*, Journal of Computational Chemistry, vol. 14, No. 11, 1385-1395, John Wiley & Sons Inc., 1993.

[33]. Judson R.S. et al, *Conformational Searching Methods for small Molecules. II. Genetic Algorithm Approach*, Journal of Computational Chemistry, vol. 14, No. 11, 1407-1414, John Wiley & Sons Inc., 1993.

[34]. Kumar et al, Introduction to Parallel Computing, Redwood City, CA: Benjamin/Cummings Publishing Company Inc., 1994.

[35]. Lamont, Gary B. & Merkle, Laurence D., CSCE686 Lecture & class notes, 1995.

[36]. Lamont et al, *Genetic Algorithm Toolkit*, ver 2.0, 1993.

[37]. Larranaga et al, Genetic Algorithms Elitist Probabilistic of Degree 1, a generalization of Simulated Annealing, University of the Basque Country, June 21,1993.

[38]. LeGrand, Scott M. & Merz, Kenneth M. Jr., *The Application of the Genetic Algorithm to the Minimization of Potential Energy Functions*, Journal of Global Optimization, pp. 49-63, 1993.

[39]. Lengauer, Thomas, *Algorithmic Research Problems in Molecular Bioinformatics*, Arbeitspapiere der GMD 748, May 1993.

[40]. Lin, Feng-Tse, et al, *Applying the Genetic Approach to Simulated Annealing in Solving Some NP-Hard Problems*, IEEE Transactions on Systems, Man, and Cybernetics, vol. 23, no. 6, pp. 1752-1767, 1993.

[41]. Meijer, Anton & Peeters, Paul, Computer Network Architectures, Rockville, MD: Computer Science Press, 1983.

[42]. Merkle, Laurence D. & Gates, George H. Jr., Series of personal conversations, June 1995.

[43]. Merkle, Laurence D., Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms, MS thesis, AFIT/GCE/ENG/92-D, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1992.

[44] Merkle, Laurence D., Prospectus on Optimal Parameter Selection for a Class of Evolutionary Algorithms, Technical Report, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, April 21, 1995.

[45]. Merkle, Laurence D., Gaulke, Robert L. et al., *Hybrid Genetic Algorithms for Polypeptide Energy Minimization*, paper submitted to Symposium on Applied Computing '96, September 15, 1995.

[46]. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, NY, NY: Springer-Verlag, 1992.

[47]. National Science and Technology Council, High Performance Computing and Communications: Technology for the National Information Infrastructure, Supplement to the President's Fiscal Year 1995 Budget, Second Printing.

[48]. Nutt, Gary, Open Systems, Englewood Cliffs, NJ: Prentice Hall Inc., 1992.

[49]. Orvosh, D & Davis, L., *Shall We Repair? Genetic Algorithms, Combinatorial Optimization, and Feasibility Constraints*, Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, p. 650, 1993.

[50]. Pearl, Heuristics - Intelligent Search Strategies for Computer Problem Solving, Reading, MA: Addison-Wesley Publishing, 1984.

[51]. Press, William H. et al, Numerical Recipes in C, Cambridge, MA: Cambridge University Press, 1990.

[52]. Sandia National Laboratories, *Sandia/Intel Set World SuperComputing Speed Record—Again*, Albuquerque, NM, Jan 1995.

[53]. Schulze-Kremer, Steffen, *Genetic Algorithms for Protein Tertiary Structure Prediction*, Parallel Genetic Algorithms, pp. 129-49, ISO Press, 1993.

[54]. Singhal, Mukesh & Shivaratri, Niranjan G., Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems, NY, NY: McGraw-Hill Incorporated, 1994.

[55]. Strang, Gilbert, Introduction to Applied Mathematics, Wellesley, MA: Wellesley-Cambridge Press, 1986.

[56]. Thompson, H. Bradford, *Calculation of Cartesian Coordinates and Their Derivatives from Internal Molecular Coordinates*, The Journal of Chemical Physics, vol. 47, no. 9, pp. 3407-10, November, 1967.

[57]. White, Handler, Smith, Principles of Biochemistry, fourth ed., NY, NY: McGraw-Hill Book Company, 1968.

[58]. Whitley, Darrell & Gordon, Scott, & Mathias, Keith, *Lamarckian Evolution, The Baldwin Effect and Function Optimization*, Parallel Problem Solving from Nature - PPSNIII, Davidor, Yuval et al (Eds.), International Conference on Evolutionary Computation, Berlin: Spring-Verlag, pp. 6-15, October 9-14, 1994.

[59]. Winston, Artificial Intelligence, Third Ed, Reading, MA: Addison-Wesley Publishing, 1993.

[60]. Yong, Liu & Lishan, Kang & Evans, D.J., *The annealing evolution algorithm as function optimizer*, Parallel Computing, Vol. 21, pp. 389-400, 1995.

INDEX

Vita

Captain Robert L. Gaulke earned his bachelor's degree in Computer Science and Mathematics from the University of Tampa in 1991. He earned his commission through the Air Force Reserve Officer Training Corps (AFROTC). Upon entering active duty in 1992, he was assigned to the Air Force Military Personnel Center (AFMPC), Directorate of Personal Data Systems, Modeling and Retrieval Section. While there, he served in a customer service and training capacity for the central-site personnel data systems worldwide. He left AFMPC in 1994 to attend the Air Force Institute of Technology.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	08 Dec 95	Master's Thesis

4. TITLE AND SUBTITLE
The Application of Hybridized Genetic Algorithms to the Protein Folding Problem

5. FUNDING NUMBERS

6. AUTHOR(S)
Robert L. Gaulke, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/95D-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Wright Laboratory (AFMC)
Materials Directorate
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The protein folding problem consists of attempting to determine the native conformation of a protein given its primary structure. This study examines various methods of hybridizing a genetic algorithm implementation in order to minimize an energy function and predict the conformation (structure) of [Met]-enkephalin. Genetic Algorithms are semi-optimal algorithms designed to explore and exploit a search space. The genetic algorithm uses selection, recombination, and mutation operators on populations of strings which represent possible solutions to the given problem. One step in solving the protein folding problem is the design of efficient energy minimization techniques. A conjugate gradient minimization technique is described and tested with different replacement frequencies. Baldwinian, Lamarckian, and probabilistic Lamarckian evolution are all tested. Another extension of simple genetic algorithms can be accomplished with niching. Niching works by de-emphasizing solutions based on their proximity to other solutions in the space. Several variations of niching are tested. Experiments are conducted to determine the benefits of each hybridization technique versus each other and versus the genetic algorithm by itself. The experiments are geared toward trying to find the lowest possible energy and hence the minimum conformation of [Met]-enkephalin. In the experiments, probabilistic Lamarckian strategies were successful in achieving energies below that of the published minimum in QUANTA.

14. SUBJECT TERMS
Genetic Algorithms, Protein Folding Problem, Niching, Lamarkian Evolution, Baldwinian Evolution

15. NUMBER OF PAGES
130

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.